Sound Tools ERS

Revision 2.1

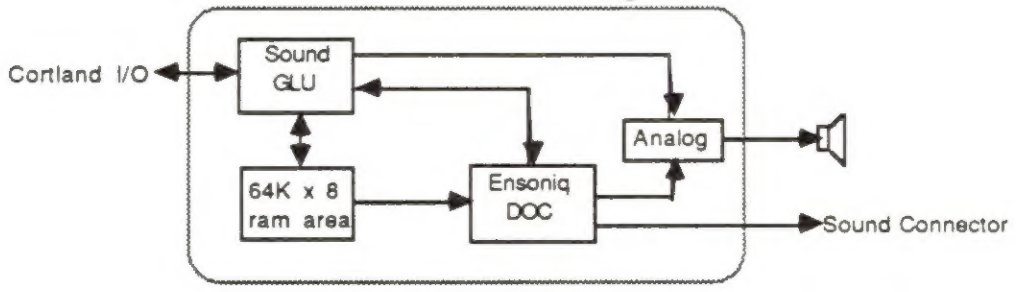
March 2, 1987

1.0 Introduction.

The sound tool package gives developers the ability to access the Sound hardware without having to know specific hardware I/O addresses. The Cortland sound hardware comes in two configurations. The first configuration is 100% compatible with the Apple //e sound capabilities. In this mode applications toggle a soft switch, which in turn generates clicks in a speaker. Also, with Cortland it is possible for an application to control the volume of the speaker.

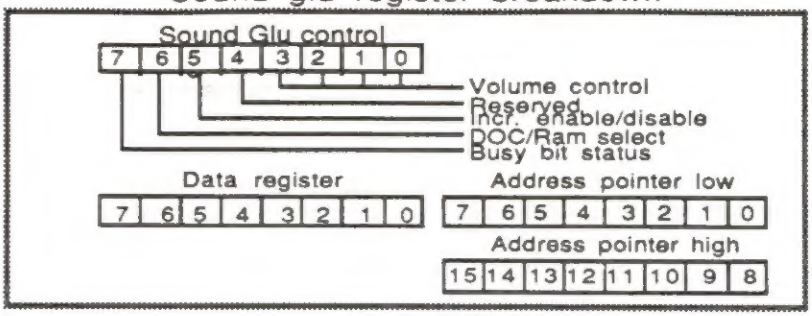
The second configuration requires the Ensoniq (DOC) digital oscillator chip and two 64K x 4 ram chips. The sound tools will contain all of the firmware routines required to access the hardware in the Ensoniq configuration. The following block diagram shows the major functional blocks of the sound hardware.

Cortland Sound Block diagram



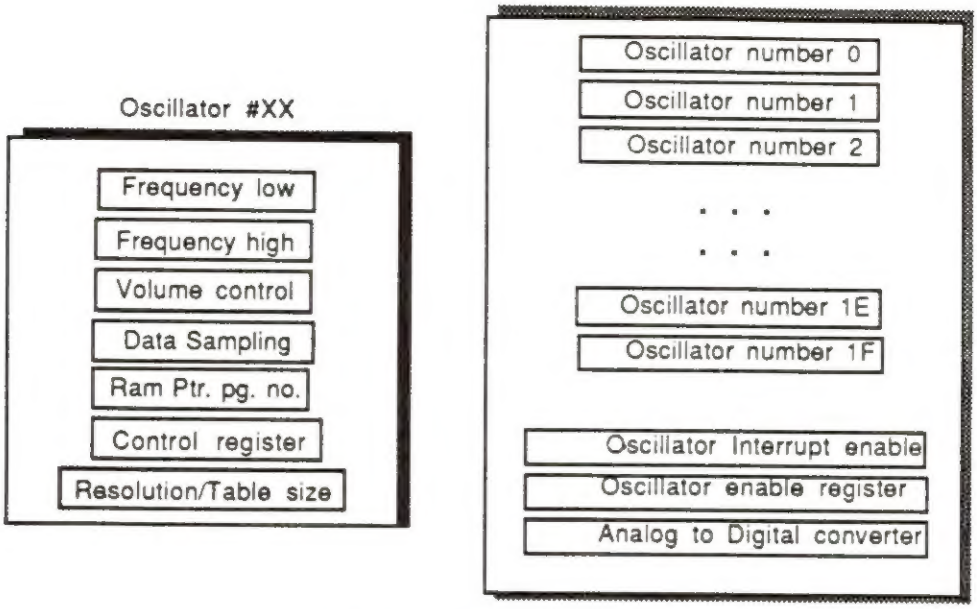
The sound GLU acts as the interface chip between the Cortland I/O and system volume, ram chips or the DOC. The following diagram shows a diagram for the register breakdown for the sound glu:

Sound glu register breakdown



The DOC ram is used to store waveforms which will be used by the DOC for sound generation. The DOC is the work horse of the sound system. With this chip we can create sounds of any pitch and duration. A register breakdown of the DOC follows:

Ensoniq Digital Oscillator Chip
Register Allocation



DOC register table

REG#	Function	D7	D6	D5	D4	D3	D2	D1	D0
00-1F	FREQ. LOW	FL7	FL6	FL5	FL4	FL3	FL2	FL1	FL0
20-3F	FREQ. HI	FH7	FH6	FH5	FH4	FH3	FH2	FH1	FH0
40-5F	VOLUME	V7	V6	V5	V4	V3	V2	V1	V0
60-7F	DATA SAMPLE	W7	W6	W5	W4	W3	W2	W1	W0
80-9F	WAVEFORM TABLE PTR	P7	P6	P5	P4	P3	P2	P1	P0
A0-BF	CONTROL	CA3	CA2	CA1	CA0	1E	M2	M1	H
C0-DF	BANK SEL/TBL. SIZE/RES.	X	BS	T2	T1	T0	R2	R1	R0
E0	OSCILLATOR INTERRUPT	IRQ	1	04	03	02	01	00	1
E1	OSCILLATOR ENABLE	X	X	E4	E3	E2	E1	E0	X
E2	A/D CONVERTER	S7	S6	S5	S4	S3	S2	S1	S0

Please refer to the Ensoniq DOC Ers for a detailed description of the part.

The analog section contains all the circuitry needed to amplify and filter the signal coming from the Sound Glu or the DOC, which will be sent to the speaker.

Finally, the sound connector gives developers the ability to design interface cards, which can take the tones generated by the DOC and modify them further. Two examples of possible sound cards are, programmable

filter stereo interface cards, and sound sampling cards. The remainder of this document will deal with a detailed description of the Sound tool calls and how they can be used to access the hardware to generate sounds.

1.1 Sound Tools Definitions.

An *oscillator* is defined as the basic sound generating unit in the DOC. The DOC contains thirty two oscillators; each of which can function independently from all the other oscillators.

One of the modes the DOC can be set to is called swap mode. In this mode each pair of oscillators is grouped together to form a swap pair of oscillators. This is the mode used by the Free Form synthesizer to generate sounds. Each of these swap pair of oscillators is called a *Generator*. An oscillator to generator translation table has been defined to get the generator number corresponding to a particular oscillator number. There are fifteen generators defined in the Cortland sound system. Oscillators thirty and thirty one a reserved for use by Apple Computer and should not be used by application programs. If an interrupt is generated by oscillator thirty of thrity one control will be passed to the system death manager with a message "Unclaimed sound interrupt". Please note, when the SoundBootInit Call is made, all of the sound interrupt handler pointers point to the System death manager with the "Unclaimed sound interrupt message".

Before a generator can be accessed, a sound tools startup call must be made. This call assigns a work area for the sound tools. The work area is broken down into sixteen groups of sixteen bytes each. Each sixteen byte group is defined to be a *generator control block (GCB)*. The first byte of each GCB is defined to contain the synthesizer mode being used by that generator. The low nibble of the byte contains the mode. The high nibble is reserved for use by the system. The remaining fifteen bytes are user definable.

The Sound tools set is made up of four main blocks; the Free Form Synthesizer, the Note Synthesizer, the Note Sequencer and the Instrument generator.

1.2 Free form synthesizer tool set definition.

As mentioned before, the tool set gives a developer the ability to control the sound hardware without having to access the hardware registers directly. The tool set is defined from the point of view of a complete sound system. The tool set must be able to read and write to ram, read and write to the DOC registers and raise and lower the volume.

The sound tool package is accessed through the Tool locator. This tool locator lets an application set up parameter lists on the stack, call tool functions and return to the caller with return parameters on the stack. It is the responsibility of the caller to make room on the stack for values which may be returned to the caller from the tool calls.

The Sound Tool set has a tool number assigned to it. With this tool number the Tool Locator can access the sound tools.

The sound tool calls are broken down into two groups. The first group of calls is made through the Tool Locator. Each of these calls has a function number assigned to it. With this function number the Tool locator knows which function to call within the tool set. All parameters for these calls are passed on the stack. Function results are returned to the caller in the stack. The number of parameters will vary depending on the type of call being made. It is the responsibility of the individual tool functions to do the stack manipulation to keep it aligned. Also the accumulator and the carry bit will reflect the success or failure of the function call. Please refer to the "Tool locator" documentation for a detailed description of the interface.

The second group is a set of routines which can be accessed through a jump table located somewhere in ram. Parameters are passed to these

routines in the processors registers. Results from these calls are passed back in registers. The following list gives a breakdown of the sound tools.

Sound tool function calls:

Group A (Function calls)		Group B (Low level routines)**
SoundBootInit	= $01	Read Register
SoundStartup	= $02	Write register
SoundShutdown	= $03	Read Ram
SoundVersion	= $04	Write Ram
SoundReset	= $05	Read Next
SoundToolStatus	= $06	Write Next
WriteRamBlock	= $09	Disable Incr.
ReadRamBlock	= $0A	
GetTableAddress	= $0B	
GetSoundVolume	= $0C	
SetSoundVolume	= $0D	
FFStartSound	= $0E	
FFStopSound	= $0F	
FFSoundStatus	= $10	
FFGeneratorStatus	= $11	
SetSoundMIRQV	= $12	
SetUserSoundIRQV	= $13	
FFSoundDoneStatus	= $14	

** The low level routines are entered through a jump table. The table address can be obtained through a call to "Get Address" function. The format of the jump table is as follows:

Offset				
Read Register $00	Addr low	Addr high	Bank	$00
Write Register $04	Addr low	Addr high	Bank	$00
Read Ram $08	Addr low	Addr high	Bank	$00
Write Ram $0C	Addr low	Addr high	Bank	$00
Read Next $10	Addr low	Addr high	Bank	$00
Write Next $14	Addr low	Addr high	Bank	$00
Osctable $18	Addr low	Addr high	Bank	$00
Generator table $1C	Addr low	Addr high	Bank	$00
Gcb.addr. table $20	Addr low	Addr high	Bank	$00
Disable incr. $24	Addr low	Addr high	Bank	$00

SoundBootInit **function #01**

This call is made on system powerup or system reset to bring the sound hardware to powerup state. The call is made by the firmware and can NOT be made by an application program! This call will reset all of the DOC sound memory to $80, zero out the sound tools work areas, halt all the oscillators and turn the volumes down to zero.

Error Codes: None

SoundStartup **function #02**

The Sound tools startup call is made by an application to set up a sound tools work area. This call must be the first call made by the application program. The call initializes a work area to be used by the sound tools. The pointer to the work area must be passed as a parameter to the call. This work area will be used as a zero page. This page will be allocated by calling the memory manager. It must be page aligned and locked until a shutdown call is made. The stack configuration for the call is as follows:

```
Stack configuration for SAppInit

Wap:word ; Work area pointer in Bank $00
```

Error Codes:

$0810 = No DOC chip or ram found.
$0818 = Sound tools already started

Example:

```
PEA Label                ; One page work area in bank $00
_SoundStartup            ; Sound Tools startup macro call
```

SoundShutdown **function #03**

This call will shut down the sound tools. It shuts off all of the oscillators resets the WAP back to $0000 and zeros out the sound tools work memory to zero. There are no parameters passed to the call on the stack and no values returned It is the responsibility of the application to release the memory allocated to the work area back to the memory manager.

Error Codes: None

Example:

```
_SoundShutDown          ; shutdown the free-form sound tools
```

SoundVersion **function #04**

This call returns the Sound tools version number. The format of the version number is as specified in the Tool Locator documentation. There are no parameters passed to the call but room must be made on the stack for one word of version information returned to the caller.

Error Codes: None

Example:

```
PEA $0000              ; make room for version
_SoundVersion          ; Sound Tools version call
```

SoundReset **function #05**

This call stops all of the generators which may be generating sound. This call can not be used by an application to stop sound generation. It is intended for use by the firmware to control the shutdown of generators. An application program should use the stop sound call to shut down a generator. This call does not require any parameters on the stack or returns any values back to an application. This call does not update the active generators flag.

Error Codes: None

SoundToolStatus **function #06**

This call will return the status of the sound tools. It returns a $FFFF if a SAppInit ($02) call has been made; otherwise it returns $0000. Room must be made on the stack for a one word value which will be returned to the caller.

Error Codes: None

Example:

```
PEA $0000              ; make room For sound tools status
_SoundToolStatus       ; Sound Tools Started status
```

WriteRamBlock **function #09**

The Write Ram Block call will write a specified number of bytes from system ram into DOC ram. The parameter list is made up of the starting address, and a byte count to move. If the sum of the starting address and the byte count are greater than 64K, an error status will be returned. Please note that if your source buffer includes the address range for I/0 space of bank $00, bank $01 and bank $E0 or bank $E1 if shadowing is on, you will access soft switches which will crash the system.

Stack configuration for write ram block:

```
Source_ptr:Long word     ; data source start address
DOC.start:word           ; DOC buffer start address
Byte_count:word          ; number of bytes to move
```

Error Codes:
$0810 = No DOC chip or ram found.
$0811 = DOC address range error.

Example:

```
Pushlong Label           ; Source buffer address
PEA DOC.buff             ; DOC ram buffer start address
PEA byte.count           ; number of bytes to move
_WriteRamBlock           ; Write ram block macro call
```

ReadRamBlock **function #0A**

This call reads any number of locations from the 64K DOC ram area into a user specified buffer. The number of bytes and the starting location must not add up to a value greater than 64K, otherwise a range error will be generated. It is the responsibility of the application to make sure that the memory manager has been called to allocate a buffer to deposit the data read from the DOC ram. The format of the parameter list is as follows:

Stack configuration for Read Ram block

```
Dest_ptr:Long word      ; Destination system buffer address
DOC.start:word          ; Source start address in DOC ram.
Byte_count:word         ; number of bytes to move
```

Error Codes:
$0810 = No DOC chip or ram found.
$0811 = DOC address range error.

Example:

```
Pushlong Label          ; System ram buffer start address
PEA DOC.buff            ; DOC ram buffer start address
PEA byte.count          ; number of bytes to move
_ReadRamBlock           ; Read ram block macro call
```

GetTableAddress **function #0B**

This call returns the jump table address for the fast access routines. The table of low level routines is defined as follows:

	Offset				
Read Register	$00	Addr low	Addr high	Bank	$00
Write Register	$04	Addr low	Addr high	Bank	$00
Read Ram	$08	Addr low	Addr high	Bank	$00
Write Ram	$0C	Addr low	Addr high	Bank	$00
Read Next	$10	Addr low	Addr high	Bank	$00
Write Next	$14	Addr low	Addr high	Bank	$00
Osctable	$18	Addr low	Addr high	Bank	$00
Generator table	$1C	Addr low	Addr high	Bank	$00
Gcb.addr. table	$20	Addr low	Addr high	Bank	$00
Disable incr.	$24	Addr low	Addr high	Bank	$00

With the exception of the Osctable, Generator table and GCB addr. table each of these routines are defined later in this document.

The Osctable translates from generator number to oscillator number, . The oscillator number returned through this table is the first oscillator of the pair. The Gcb address table points to the first location of the GCB corresponding to a generator, and the Generator table translates from oscillator number to generator number.

The application making this call must make room on the stack for a long word returned from the call.

Error codes:None

Example:

```
Pushlong $00000000      ; Make room for long address
_GetTableAddress        ; Get table address macro call
```

GetSoundVolume **Function $0C**

This call will read the volume setting for a generator. The possible range of values read back are between $00-$FF. All eight bits are valid for DOC volume registers.

If the generator specified is greater than fourteen ($0E), then the system volume setting will be returned. The hardware for the system volume control uses the low nibble of a byte to set the volume. In order to be consistent with the DOC volume registers, we map the low nibble into the upper nibble of a byte. We end up with each possible system volume setting mapped sixteen times. Volume settings $00-$0F correspond to system volume setting $00, values $10-$1F correspond to system volume $01, etc.

Room must be made on the stack for a one word value which will be returned from the call.

```
Stack configuration for Get Volume call:
Gen_number:word          ; Generator number

Error codes:None

Example:
        PEA $0000        ; room for volume setting
        PEA gen.num      ; Generator number
        _GetSoundVolume  ; Get volume macro call
```

SetSoundVolume **Function $0D**

The set volume call changes the volume setting for the volume registers in the DOC and the system volume. Generator numbers $00-$0E will set the volume on pairs of generators in the DOC. Generator numbers $0F or greater will set the system volume control. The range of values for the volume setting are $00-$FF. The DOC volume registers use all eight bits of resolution. The system volume control will use the upper nibble of the setting to determine the setting.

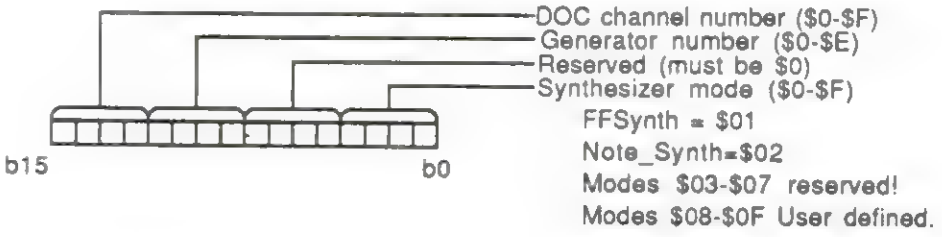
```
Stack configuration for SetVolume call:
Volume_setting:word     ; new volume setting
Gen_number:word         ; Generator number to  set

Error codes: None

Example:
    PEA New.volume      ; new volume setting
    PEA gen.num         ; Generator number
    _SetSoundVolume     ; Get volume macro call
```

FFStartSound **Function $0E**

This call will enable the DOC to start generating sound on a particular generator based on the parameter list passed to the call. If a generator is already active and a start sound call is made for it, then the previous sound generation process will be terminated and the new sound process will be started. The parameter list for the Start Sound call is as follows:

The stack configuration for StartSound call:

GenNumb./FFsynth:word ; Channel no./generator number/mode




```
Pblock_ptr:Longword          ; Parameter block pointer
```

The parameter block format:

```
Wave_start:Longword          ; Start address of wave
Wave_size:word               ; Waveform size in pages[1]
Freq_offset:word             ; waveform playback frequency[2]
DOC_buffer:word              ; DOC buffer start address[4]
DOC_buffer_size:word         ; DOC buffer size code[3]
Nextw_addr:Longword          ; Next wave parameter block ptr[5]
Volume_setting:word          ; DOC volume setting.[4]
```

1. The smallest which can be played back is one page. A waveform size of $FFFF will play back 65536 pages.
2. The Frequency register setting can be calculated with the following formula: FR=((32*PF)/1645), where PF=Playback frequency in hertz & FR=Frequency register value.
3. This code assigns a size for the DOC buffer used for the waveform being played. One of these buffers is assigned for each oscillator in the generator pair being used to play the waveform. The DOC start address for the second oscillator is assigned at start address + DOC buffer size.

4. For further information on these settings, please refer to the Ensoniq DOC ERS.
5. These three bytes point to another waveform parameter block. If the setting of the Nextw_addr and Nextw_bank are zero, then there are no more Free-Form synthesizer waveforms to be played back through this start sound call.

Error Codes:
$0812 = NO SAppInit call made
$0813 = Invalid generator number
$0814 = Synthesizer mode error
$0815 = Generator busy
$0817 = Master IRQ not assigned

Example:

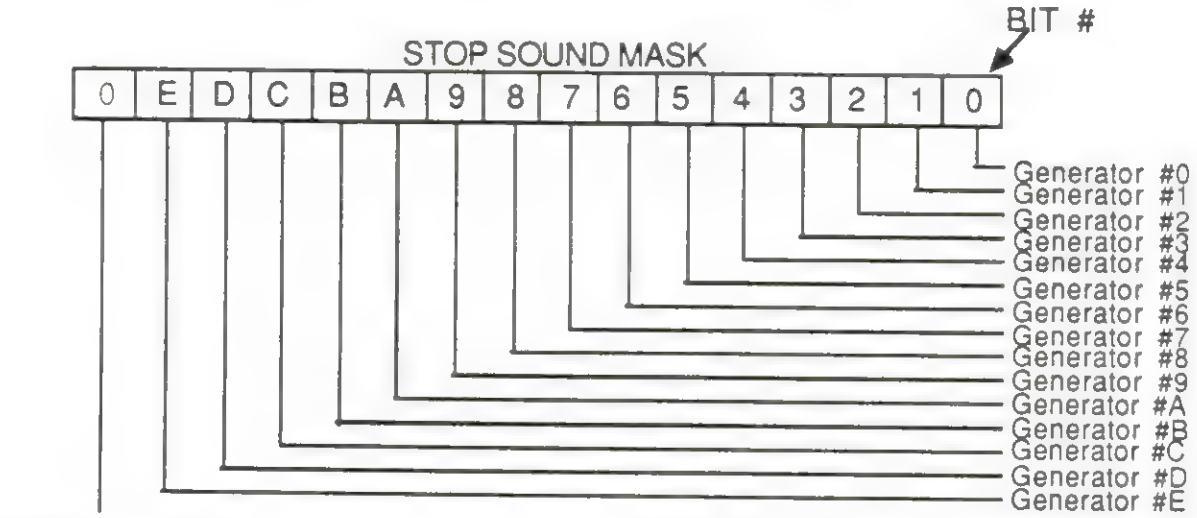
```
        PEA Gen.mode            ; Generator/mode word
        Pushlong Pblock         ; Parameter block pointer
        _FFStartSound           ; Free Form Synth start sound macro
            . . . .
            . . . .
            . . . .
Pblock equ *                    ; Waveform parameter block
        DC  I4'Wave.start'      ; Waveform start address
        DC  I2'Wave.size'       ; Wave size in pages (1 page min.)
        DC I2'DOC.Freq'         ; DOC frequency register value
        DC  I2'DOC.buffer'      ; DOC ram buffer start address
        DC I2'DOC.buf.code'     ; DOC buffer size code $00-$07
        DC  I4'Next.wave'       ; next wave parameter block ptr.
        DC I2'DOC.volume'       ; DOC volume register setting
        . . . .
        . . . .
        . . . .
Next.wave equ *                 ; Next wave parameter block
        . . . .
        . . . .
        . . . .
```

FFStopSound **Function $0F**

This call will stop sound generators which may be running. A generator running is defined to be one playing a waveform or one which has completed playing a waveform. The generator will stay busy until a stop sound call is made, even though waveform playback has ended. Depending on the setting of a sixteen bit flag passed as a parameter to the function any of fifteen generators will be stopped if running. Each bit position in the stop generator mask corresponds with a sound generator. Bit zero corresponds to generator zero, bit one corresponds with generator number one, and so on. There are only fifteen generators defined. This call does not return any error information back to the caller. The format of the parameter list is as follows:

Stack configuration for Stop Sound:

```
Gen_mask:word          ; generators to stop
```



Must be zero

1 = stop corresponding generator
0 = leave it generator alone

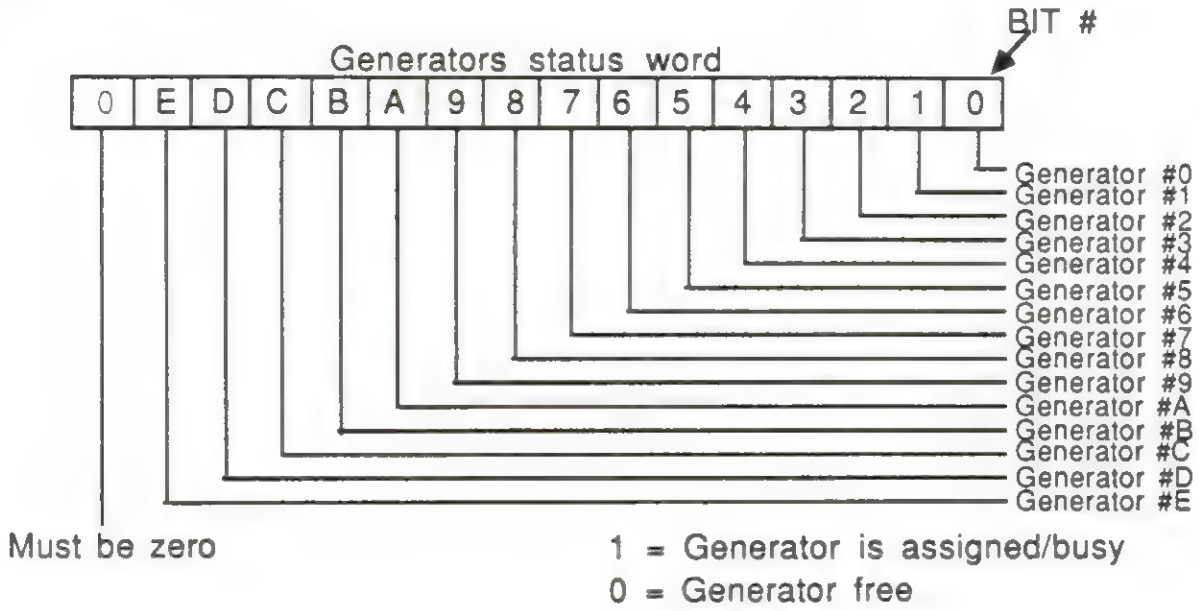
Error Status: None

Example:

```
PEA Stop.mask          ; mask for stop generators
_FFStopSound           ; Free Form Synth stop sound
```

FFSoundStatus **Function $10**

This call will return the status of the all fifteen generators. Any bit position in the status word returned from the function call signifies that the corresponding generator is active. There are no parameters passed to the function. The format of the word returned form the call is as follows:



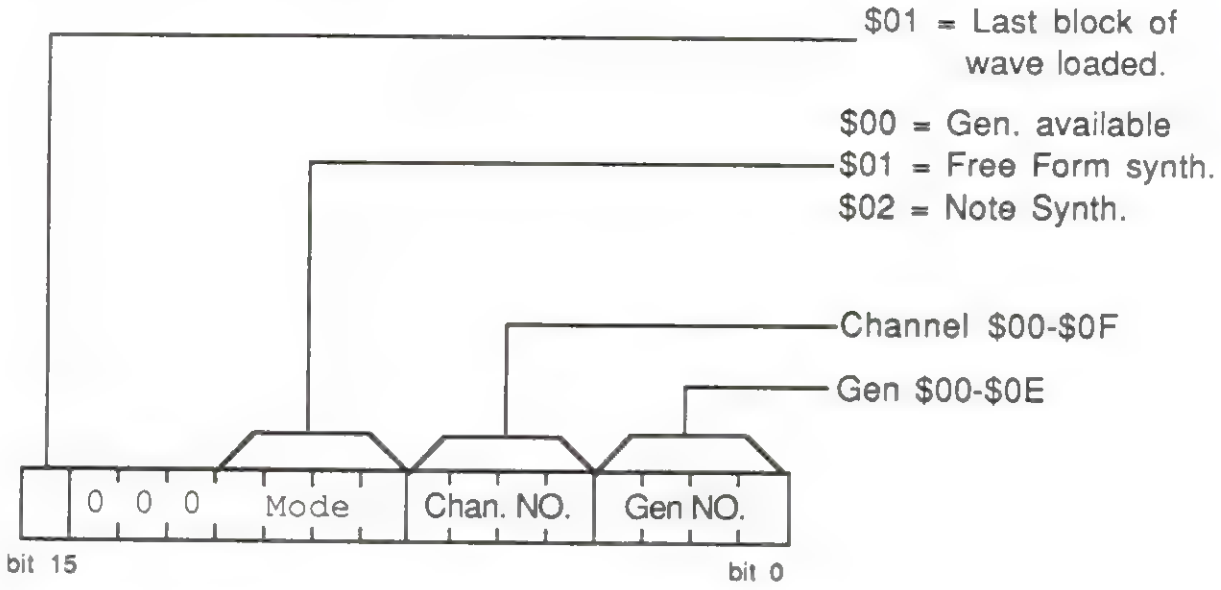
Error Status: None

Example:

```
PEA $0000              ; make room for status word
_FFSoundStatus         ; Generators status macro call
```

FFGeneratorStatus **Function=$11**

This call will read the first two bytes of the GCB corresponding to the generator specified. Room must be allocated on the stack for the word returned from the call. For the Free form synthesizer these two bytes have the following format:



Stack configuration for Gen. status call:

```
Gen_number:word          ; generator number for status
```

Error Codes: None

Example:

```
PEA $0000                ; room for Generator status
PEA Gen.number           ; Generator number
_FFGeneratorStatus       ; generator status macro call
```

SetSoundMIRQV **Function=$12**

This calls sets up the entry point into the sound interrupt handler. This routine will be accessed every time an interrupt is generated by the DOC. The processor will be in full native mode when the sound interrupt handler is entered. The parameter list for a set sound IRQ vector is as follows:

```
Set Master Sound Irq vector stack config.
SMaster_irq:Longword     ; Sound Master IRQ vector

Error Codes: None
```

Example:

```
Pushlong Master.irq.vect ; Set master irq vector macro
_SetSoundMIRQV           ; set the master sound vector
```

SetUserSoundIRQV **Function=$13**

This calls sets up the entry point for a users synthesizer interrupt handler. When an interrupt occurs for a user defined synthesizer then control will be passed to the ram based synthesizer code through this vector. The old vector installed will passed back to the caller. This old vector must be preserved by the caller. If control is passed to the user vector and the synthesizer mode is not his, then control will passed further down the chain through this vector. Control will be passed through a JSL, therefore the user must return control through an RTL instruction. Room must be made on the stack for long word returned on the stack.

```
Stack configuration for Set User's Sound IRQ vector.
User_irq_vector:Longword                ; New user IRQ vector
```

Error Codes: None

Example:

```
Pushlong $00000000      ; make room for old vector
Pushlong New.vector     ; new vector
_SetUserSoundIRQV       ; set user sound irq vector macro
```

FFSoundDoneStatus **Function #14**

This call will return the status of the Free Form synthesizer sound playing status. If the generator specified is currently playing out a waveform, then the status returned to the caller will be $0000. If the generator is done playing then the status will be $FFFF. Room must be made on the stack for one word of status returned to the caller.

```
Stack configuration for FFSoundDoneStatus
Gen_number:word          ; Generator number

Error codes:
$0813 = Invalid generator number
```

Example:

```
PEA $0000                ; Make room for status
PEA Gen.number           ; Generator number to check
_FFSoundDoneStatus       ; FFsynth Sound done stat.  macro
```

Read register**

This low level routine lets an application read any DOC register. The routine is entered through the Jump table provided by the "GetTableAddress" function call. This call will return to the caller through an RTL instruction. After this call is made the Sound Glu register is left in register access mode with auto increment enabled.

Through the generator to "oscillator" table, an application can ascertain the setting of any register corresponding to an oscillator.

Import:
e = 0 ; native mode
m = 1; 8 bit accumulator
x = 0 ;16 bit index registers
X = DOC register to read

Export:
AL = contents of register requested

Error codes: None

DOC register table

REG#	Function	D7	D6	D5	D4	D3	D2	D1	D0
00-1F	FREQ LOW	FL7	FL6	FL5	FL4	FL3	FL2	FL1	FL0
20-3F	FREQ HI	FH7	FH6	FH5	FH4	FH3	FH2	FH1	FH0
40-5F	VOLUME	V7	V6	V5	V4	V3	V2	V1	V0
60-7F	DATA SAMPLE	W7	W6	W5	W4	W3	W2	W1	W0
80-9F	WAVEFORM TABLE PTR	P7	P6	P5	P4	P3	P2	P1	P0
A0-BF	CONTROL	CA3	CA2	CA1	CA0	1E	M2	M1	H
C0-DF	BANK SEL/TBL. SIZE/RES.	X	BS	T2	T1	T0	R2	R1	R0
E0	OSCILLATOR INTERRUPT	IRQ	1	04	03	02	01	00	1
E1	OSCILLATOR ENABLE	X	X	E4	E3	E2	E1	E0	X
E2	A/D CONVERTER	S7	S6	S5	S4	S3	S2	S1	S0

Note: Register types are grouped into register classes. Within each register class, the register number for each oscillator is assigned in ascending order. For example: the low byte of the frequency register for oscillator zero is register $00, the low byte of the frequency for oscillator number is register $01. The high frequency register for oscillator number zero is accessed through register number $20, oscillator one uses register number $21 etc... The register numbers are provided in the table defined above.

Write register **

The Write DOC call will write a one byte parameter to any register in the DOC chip. The call will be made through the jump table provided to the application by the tool call "Get Address". To write to an oscillator register corresponding to a generator we get the oscillator number from the oscillator table, bump it by one if we want to access the odd oscillator of the pair, add the base register of the specific register we want to access and then make the write register call through the Write register routine address in the jump table. This call will return to the caller through an RTL instruction. After this call is made the Sound Glu register is left in register access mode with auto increment enabled. Please refer to the "Note" in the Read register description for information on register assignments for each oscillator.

Import:
e = 0 ; native mode
m = 1; 8 bit accumulator
x = 0 ; 16 bit index registers
AL = data to write
X = DOC register number

Error codes: None

DOC register table

REG#	Function	D7	D6	D5	D4	D3	D2	D1	D0
00-1F	FREQ. LOW	FL7	FL6	FL5	FL4	FL3	FL2	FL1	FL0
20-3F	FREQ. HI	FH7	FH6	FH5	FH4	FH3	FH2	FH1	FH0
40-5F	VOLUME	V7	V6	V5	V4	V3	V2	V1	V0
60-7F	DATA SAMPLE	W7	W6	W5	W4	W3	W2	W1	W0
80-9F	WAVEFORM TABLE PTR	P7	P6	P5	P4	P3	P2	P1	P0
A0-BF	CONTROL	CA3	CA2	CA1	CA0	1E	M2	M1	H
C0-DF	BANK SEL/TBL. SIZE/RES.	X	BS	T2	T1	T0	R2	R1	R0
E0	OSCILLATOR INTERRUPT	IRQ	1	04	03	02	01	00	1
E1	OSCILLATOR ENABLE	X	X	E4	E3	E2	E1	E0	X
E2	A/D CONVERTER	S7	S6	S5	S4	S3	S2	S1	S0

Read Ram**

This call will read any Ensoniq ram location specified by the caller. This call leaves the address pointer register in the Sound Glu in auto increment mode and in ram access mode. The call does not do any type of error checking on the address, or data. This call exits back to the caller through an RTL instruction. After this call is made the Sound Glu register is left in RAM access mode with auto increment enabled.

Import:
e = 0 ; native
m = 1 ; 8 bit accumulator
x = 0 ; 16 bit index registers
X = Ensoniq ram address to read

Error codes: None

Write Ram**

This call will write a one byte value to any Ensoniq ram location specified. The call does not do any type of error checking on the address or data value to be written. This call returns to the caller through an RTL instruction. After this call is made the Sound Glu register is left in RAM access mode with auto increment enabled.

Import:
e = 0 ; native
m = 1 ; 8 bit accumulator
x = 0 ; 16 bit index registers
AL = data value to be written
X = Ensoniq ram address to write to

Error codes: None

Read Next**

This call will read the next location pointed to by the Sound Glu address register. The previous call must have been a Read register, write register, read ram, or a write ram call for this call to work properly. Any of these four calls will leave the Sound Glu set to auto increment and pointing to DOC register or ram access mode. After the read is made the Sound Glu address/DOC register pointer will be incremented to the next location.

Import:

None

Export:
AL = data byte read

Error codes: None

Write Next**

This call will write one byte of data to the next DOC register or ram location depending on the setting of the Sound Glu control register. The call will write to DOC registers or ram and then increment the address pointer register in the Sound Glu, if the address pointer register was enabled for auto increment. If a Read register, read ram, write register or write ram call is made then that call will leave the Sound Glu control register in that type of access mode and with auto increment enabled.

Import:
AL = byte value to be written

Error codes: None

Disable Incr. **

This call will disable the auto increment mode set up by read ram, write ram, read reg. or write reg. By disabling the auto increment bit an application program can read a DOC register or memory location continuously. This can be useful when reading the analog to digital converter. As an example to read the AtoD register a program would make a read register call to register $E2, followed by a Disable Increment call. For each subsequent read to the analog to digital converter the program would call read next. This mode will stay in effect until a read register, write register, read ram or a write call is made.

Import: NONE
Error codes: NONE

1.2.1 Error code summary.

$0810 = No DOC chip/DOC ram found.
$0811 = Address range error.
$0812 = Sound Tools not started error.
$0813 = Invalid Generator number.
$0814 = Synthesizer mode error.
$0815 = Generator already in use (busy) error.
$0817 = Master interrupt not assigned error.
$08FF = Unclaimed sound Interrupt error.

NOTE: Error $08FF reported through the System Death Manager.

Note Synthesizer ERS

Revision 00:50

September 25, 1986

GENERAL

The Note Synthesizer is a ram-based tool set, number (decimal) 25. It provides a way of making complex musical sounds on the Cortland when it is equipped with the ENSONIQ Digital Oscillator Chip (DOC).

An application program will use the note synthesizer by making tool calls at the beginning and at the end of each note to be played. The first call is to allocate one of the sound generators. Then the specified DOC generator will be set up to produce sound with a NoteOn call. During the course of the note the DOC registers for that generator will be automatically updated on a regular basis to create the shape of the sound. This happens from a timer interrupt routine which is part of the note synthesizer. The end of a note, called the release, starts when a NoteOff call is made. Sometime later, when the note has died away to zero, the generator will be automatically deallocated.

Generators and GCB's

There are 32 oscillators in the DOC. Of these, 2 are reserved for use by Apple in the future. The remaining 30 are grouped into pairs, called generators. When the Note Synth starts up it grabs one generator to use as a timer. The remaining 14 generators are allocated on a priority basis as needed.

One page of bank zero memory must be assigned to the Sound Manager Tool Set when it is started up. This area is divided into 15 blocks of 16 bytes each, which are called Generator Control Blocks (GCB). The first byte of a GCB indicates which synthesizer is using that generator (if any). The definition of the other 15 bytes depends on which synthesizer is using it.
The GCB is used as a mailbox by the note synthesizer. It contains the current values of three "knobs" or controllers which may be changed by the application program. These are for pitchbend, vibrato depth, and volume. All three controls have a range of 0 to 127. After a call to NoteOn, the GCB will be set up as follows:

GCB:	SynthID	byte	= note synth ID = 2
	GenNum	byte	= [0..14]
	Semitone	byte	as specified in call
	Volume	byte	as specified in call
	Pitchbend	byte	64 = no bend
	VibratoDepth	byte	as specified in instrument
	Note Synth internal variables: 10 bytes		

Priority Allocation

Generators may be shared among various sound producing tools. Because the note synthesizer will, by far, use the most generators, and allocate them more often, the generator allocation is one of the note synthesizer functions. It is not uncommon for music to 'accidentally' request a new note when all the available generators are busy. The allocation scheme will allow the note synthesizer to steal generators from itself, but

not from other types of synthesizers.

A generator's priority may range from 0 to 128. When priority is zero, that means that that generator is not being used, and therefore free to be used. When it is 128, then that generator is locked and may not be stolen. Priorities between 0 and 127 are used by the note synthesizer to control the stealing of notes.

When a generator is allocated to be used by one of the synthesizers, the generator is given a new priority. The generator allocation function will return with the lowest priority generator.When the note synthesizer uses a generator, it automatically lowers its priority when the envelope hits the sustain portion, and again when it hits the release portion. When the note stops, it returns the generator to zero priority. Other synthesizers in the system (the free-form synth) should always get a generator with a priority of 128.

Interrupt Timer

The note synthesizer will use one oscillator as a free running timer to provide the update rate for the envelopes. This timer can also be accessed by an application program by using the hook provided in the Startup call. The update rate is usually between 60 and 200 Hz. The timer runs all the time, until a reset or a shutdown call is made.

Generators used for sound production by the note synthesizer should have their interrupts disabled in the DOC control register defined in the instrument. The Note Synth will treat all interrupts from its oscillators as timer interrupts.

An application which uses incoming MIDI will have a hard time on the Cortland if it uses tools. Most tools disable interrupts for long periods of time. The Note Synthesizer normally runs an interrupt service routine with interrupts disabled, and this routine may take several milliseconds when many notes are playing. Since MIDI can generate interrupts as often as every 333 usec, there is a problem. The application can force the note synth interrupt service routine to run with interrupts ENABLED. This is accomplished by installing a user timer routine. It will get called in the interrupt, immediately before the note synth service routine. If the user subroutine returns with irqs enabled, that state will remain throughout the note synth interrupt service routine. This should prevent loss of incoming MIDI data.

DOC Memory

It is up to the application to get the needed waveforms into DOC MEM, using the WriteRamBlock function. At no time should a zero be placed in the first 256 bytes of DOC memory. That would cause the timer oscillator to halt.

NSBootInit

Function Number:	$01
Input:	none
Output:	none
Errors:	none

Should be called only by Tool Loader.

NSStartup

Function Number:	$02	
Input:	*Update Rate*	WORD
Input:	*User Update Rtn*	LONG
Errors:	AlreadyInit = $1901	
	SoundNotInit = $1902	

The Sound Tools (Tool #8) should be started first or this startup will fail. The note synth shares a page of bank zero with the sound tools. NSStartup is necessary before using the other functions.

The Update Rate is the rate at which the envelopes and LFOs will be generated, and the rate at which the User Update Routine will get called. The rate value is in units of .4 Hertz. Some typical rates would be:

Rate: 60 Hz	use 60/.4 = 150
Rate: 100 Hz	use 100/.4 = 250
Rate: 200 Hz	use 200/.4 = 500 default

Use low rates for low overhead. Use a higher rate for smoother sounding envelopes and better sequencer timing resolution.

The User Update Routine is the address of a routine which will be called on every timer interrupt. It is intended to be used by a sequencer program. There are various ways that a sequencer can create various tempos from a fixed clock. The routine will be called will a jsl from native mode, with index and memory long, and should return with an rtl. If Interrupts have been re-enabled by the user update routine, they will remain that way throughout the note synthesizer update routine. If the user chooses to enable interrupts inside this routine, then it should be reentrant. If this argument is zero, then no routine will be called.

EXAMPLE

```
            pushword #150                ; 60 Hz update rate
            pushlong #MyRoutine
            _NSStartup
            .
            .
MyRoutine   lda >MyTimeCount
            dec a
            sta >MyTimeCount
            rtl
```

NSShutdown

Function Number:	$03
Input:	none
Output:	none
Errors:	NotInit = $1923

Shutdown turns off all generators *used by the note synth* and clears their priority. It replaces the sound IRQ vector with the one which was present before startup.

NSVersion

Function Number:	$04	
Input:	none	
Output:	*Version Number*	WORD
Errors:	None	

NSReset

Function Number:	$05
Input:	none
Output:	none
Errors:	none

This performs the same function as Shutdown.

NSStatus

Function Number:	$06	
Input:	none	
Output:	StartStatus	word ($FFFF=started,0=not)
Errors:	none	

This function returns 0 or $FFFF depending on whether the Note Synth was started yet.

AllocGen

Function Number:	$09	
Input:	RequestPriority	WORD
Output:	GenNum	WORD
Errors:	NoneAvailable = $1921	

AllocGen is a request for a sound generator. If successful, it returns a generator number, from 0 to 13. Which generator is returned is determined by the current priorities of the 14 generators. If one of the generators is free, that is, it has a priority of zero, then the first free one is returned. If none are free, then it looks for one to 'steal'. It finds the lowest priority generator. If that generator's priority is lower than, or equal to, the *RequestPriority,* then that generator is 'stolen'. If all generators are already of a higher priority, then the request fails. There is one exception; a generator with a priority of 128 is never stolen.

A fail is indicated by carry set on return. When successful, the generator is assigned a priority equal to *RequestPriority.*

DeallocGen

Function Number:	$0A	
Input:	GenNum	WORD
Output:	none	
Errors:	BadGenNum = $1922	

This function sets the named generator's priority to zero. It also makes sure that the oscillators are halted.

AllocGen and DeallocGen can be used to gain control of generators in the DOC for any of the synthesizer functions in the Cortland Tools, or even a user defined synthesizer function. The programmer can guarantee success in this allocation by always requesting the same priority. This means that there will never be a situation where a note will not sound because all the generators are busy. That is the simplest way to use the dynamic allocation. A more advanced use of priority would be to put higher priorities on bass notes and melody lines.

Note that this call is not necessary for generators that have played notes. The note synth automatically deallocates generators when their envelopes have dropped to zero.

An Error code is returned if the Generator number is greater that 13.

NoteOn

Function Number:	$0B	
Input:	*GenNum*	WORD
	Semitone	WORD
	Volume	WORD
	InstrumentPtr	LONG
Output:	None	
Errors:	AlreadyOn = $1924	

This function initiates the sounding of a note on the specified instrument.
GenNum is a generator number from 0 to 14. The *GenNum* used in the call should usually be obtained immediately prior to the call from a call to AllocGen.
The *Semitone* is specified in MIDI standard format: a value from 0 to 127, where middle C is 60.
The *Volume* parameter is also in the range of 0 to 127, and can be treated like MIDI velocity. This volume parameter is copied into the GCB. It is used as a scaler of the amplitude envelope. Each 16 steps of this parameter correspond to a 6 Db change in volume. Note that when the sum of the volume and the instantaneous envelope is less than 128, no sound will be heard because of the 48 Db dynamic range of the DOC.
The *InstrumentPtr* is a pointer to an Instrument structure, which is defined below, on page 15.

EXAMPLE

```
        pushword #0                 ;space for the GenNum
        pushword #64                ; Priority of this note: average
        _AllocGen
        pla
        sta GenNum

        pushword GenNum
        pushword Semitone
        pushword #127               ; max volume
        pushlong #Instrument        ; LONG ptr to inst definition
        _NoteOn

... wait a while
        pushword GenNum
        pushword Semitone
        _NoteOff
```

NoteOff

Function Number:	$0C	
Input:	*GenNum*	WORD
	Semitone	
	WORD	
Output:	None	
Errors:	None	

This function causes the envelope generator of the given note to go to the release stage. The release usually causes the volume to drop to zero in a short time. When the envelope reaches zero, the note will no longer be heard and the note is considered off. The generator's priority will then be set to zero, indicating that it is free.

The *GenNum* and *Semitone* should be the same ones that were specified in the corresponding NoteOn call. There are cases where the note is no longer sounding; for example, if the envelope had already dropped to zero or if the generator had been stolen to play another note. NoteOff checks to make sure that the named generator is indeed playing the named semitone.

AllNotesOff

Function Number:	$0D
Input:	none
Output:	none
Errors:	none

This function turns off all the notes that the note synthesizer is playing and returns them to zero priority. It will not shut down other generators, such as those used by the free-form synthesizer.

INSTRUMENT DEFINITION

An Instrument is a data structure which resides somewhere in Cortland memory. A NoteOn call must pass a pointer to an instrument.

Instrument:

offset	Field Name	size
0	*Envelope*	24 BYTES
24	*ReleaseSegment*	BYTE
25	*PriorityIncrement*	BYTE
26	*PitchBendRange*	BYTE
27	*VibratoDepth*	BYTE
28	*VibratoSpeed*	BYTE
29	*Spare*	BYTE
30	*AWaveCount*	BYTE
31	*BWaveCount*	BYTE
32	*AWaveList:*	AWaveCount * 6 BYTES
X	*BWaveList:*	BWaveCount * 6 BYTES

(X = 32 + AWaveCount * 6)

The *Envelope* is composed of up to eight linear segments. Each segment is described by a breakpoint and an increment. During each segment, the volume of the note will ramp from its current value to the breakpoint value. The slope, and thus the time of the ramp, is determined by the increment.

The breakpoint should be a byte between 0 and 127. It represents sound level in a logarithmic scale: each 16 steps change the amplitude by 6 Db.

The slope is described by an increment which will be added or subtracted from the current level at the update rate (100 times a second). The increment is a two byte fixed point number, that is, the lower 8 bits represent a fraction. Thus when the increment is 1 it represents 1/256. In this case, the increment would have to be added 256 times (2.56 seconds) to cause the envelope level to go up by 1.

The envelope is a list of breakpoints and increments:

stage 1:	breakpoint	increment
stage 2:	breakpoint	increment

stage 3:	breakpoint	increment
stage 4:	breakpoint	increment
stage 5:	breakpoint	increment
stage 6:	breakpoint	increment
stage 7:	breakpoint	increment
stage 8:	breakpoint	increment

Increment 1 is used to go from the initial level of 0 up to the level of breakpoint 1. Increment 2 is used to go from breakpoint 1 to breakpoint 2, and so on. The sustain level of the envelope, if there is one, is created by setting the increment to 0, causing the envelope to get "stuck" on that level. The release segment of the envelope is specified by the *ReleaseSegment* parameter, which must be a number from 0 to 7. The release may take several segments to get to zero. The last breakpoint should always be zero.

To compute the time of a segment:

$$\text{time} = \frac{|\text{last breakpoint} - \text{new breakpoint}| * 256}{\text{increment} * \text{update rate}}$$

For example, to ramp from 30 to 40, with an increment of 25, and an update rate of 100 Hz:

$$\text{time} = \frac{|30 - 40| * 256}{25 * 100\ \text{Hz}} = \frac{2560}{2500} = 1.02\ \text{seconds}$$

PriorityIncrement is a number which will be subtacted form the generator priority when the envelope reaches the sustain segment. Whe it reaches the release segment the priority will be cut in half. The proirity of each generator will also be decremented by one each time a new generator is allocated. This causes 'older' notes of to be preferred for stealing.

PitchbendRange is the number of semitones that the pitch will be raised when the 'pitchwheel' reaches 127 (the center value is 64). The valid values for PitchbendRange are 1, 2, and 4.

VibratoDepth is the (initial) fixed depth of vibrato, ranging from 0 to 127. Vibrato is a triangle shaped LFO modulating the pitch of both oscillators in a generator. A vibrato depth of zero will turn the vibrato mechanism OFF, which saves some CPU time.

VibratoSpeed controls the rate of the vibrato LFO. It can be any byte value. The actual frequency will depend of the update rate set during Startup.

AWaveCount and BWavecount tell the note synth how many waves there are in the following wavelists. There can be up to 255 waves in each list.

The **Wavelist** structure is a variable length array where each entry is 6 bytes long. Each entry represents a **Waveform**. The information is particular to the DOC; the user should refer to the DOC specification when creating instruments. Each 6 byte entry represents a waveform and contains information about the allowable pitch range of the waveform. This means that the waves can be "multi-sampled" across (an imaginary) keyboard. When a note is played, the WaveList A and B will be examined and ONE waveform will be picked out and assigned to each oscillator.

Each **Waveform** in a Wavelist has the following 6 byte format:

TopKey	byte
WaveAddress	byte
WaveSize	byte
DOCMode	byte
RelPitch	word

TopKey is the highest MIDI semitone that will be played by this waveform. The synth will examine the topkey field of each waveform until it finds one greater than or equal to the note it is trying to play. The items in the Wavelist should be in order of increasing TopKey values. The last wave in a Wavelist should have a TopKey of 127. The TopKey value is, then, the split point between waveforms.

The next three bytes will be picked up and stuffed directly into the DOC registers. The WaveAddress is the high byte of the waveform address. The WaveSize sets both the size of the wavetable and the frequency resolution. The DOC mode goes into the mode register. The interrupt enable bit will be ignored.

Briefly, some of the ways that DOCMode may be used:

Synthesizer: both oscillators, A and B, in free run mode. ($00)

Sampled, no loop: Osc A in single cycle and trigger peer mode ($06); Osc B in single cycle and halt mode, with halt set ($03). Osc A will complete and start Osc B, which will play to the end and stop.

Sampled, with loop: Osc A in single cycle and trigger peer mode ($06); Osc B in free run mode, with halt set ($01). Osc A will complete and start Osc B, which will play continuously until the note ends.

RelPitch is a two-byte word which is used to tune the waveform. This will compensate for different sample rates and waveform sizes. The high byte is in semitones, but can be a signed number. The low byte is in 1/256 semitone increments. Note that the low byte is first in memory on the 65816. A setting of

zero is the default for sounds that have one cycle per page of waveform memory.

The wavelist structure is designed to give great flexibility in creating realistic instrument timbres. It allows 'multi'sampling' with different samples of sounds on different ranges of pitch. It allows mixing of various size waveforms, with different tuning on each one. One special application might be to use a different **Waveform** entry on each semitone, to allow seperate tuning of each note. This would be a way to duplicate special tuning systems, like just temperment. The wave pointers need not be different in this case, just the RelPitch fields.

Tuning is accurate to 1/128 of a semitone in the note synth software, subject to the resolution setting of the DOC. For accurate tuning on lower notes it may be necessary to use higher settings in the DOC resolution register.

Apple IIGS Audio Compression & Expansion,

or The ACE ERS Version 01:12

"From previous experiments, I know everything reduced in size soon blows up."

Superhero Ray Palmer (**The Atom**) in Fox, Kane & Anderson's "Birth of The Atom," *Showcase* #34, DC Comics, Inc., 1961.

Table of Contents

What is ACE?

Apple's Audio Compression & Expansion Tool, ACE, is a set of utility subroutines, which you may use to compress digital audio data to half its original size or less, or to expand previously compressed data to its original size (in preparation for playback). By compressing audio data, you expedite its storage and retrieval: not only will any storage medium be able to hold at least twice as much compressed as uncompressed audio, but the effective data transfer speed between the computer's CPU and secondary storage devices, such as floppy disk or hard disk, is twice as fast for compressed as for uncompressed data. Thus, by compressing your sound samples prior to storage, you are able to compensate somewhat for limitations in a particular device's storage capacity or data access speed. The latter benefit also applies to the transmission of data over slow lines, such as those used to access commercial timesharing services. As a rule, transmitting compressed data via modem is more economical than transmitting uncompressed data.

Who Should Use ACE, and Why?

ACE was designed for use by serious amateur or professional software developers who wish to minimize the amount of storage space or data transfer time necessary to work with digitized audio samples of reasonable fidelity and length. Such minimization is desirable because of the truly vast amounts of data that are required to faithfully digitize a high-fidelity audio signal. Suppose, for instance, that you decide to digitize only thirty seconds of a song that is being broadcast by an FM radio station. Even if you record just a monophonic version of the signal, an otherwise faithful digital representation that is suitable for playback on the Apple IIGS or the Macintosh will occupy nearly a megabyte of RAM. Even assuming the signal is converted to the eight-bit samples that are used by the sound processors in Apple computers, and not the sixteen-bit samples used by compact discs or digital tape recorders, a digitized copy of an entire four-minute song would probably require more RAM than your computer can hold, especially if you record in stereo. What's more, you would need a box or two of floppy disks, or one reasonably large hard disk, in order to make a permanent copy of the digitized song.

As another example, a software publisher may wish to endow an educational package or business productivity system with a pleasing human voice: one that pronounces and inflects words more or less as a human would, without the robotic "accent" and monotonic delivery that the average person has come to associate with computer speech. Given the current state of voice synthesis technology, such a feat would require—at very least—the digitization of an actual human voice as it spoke an appropriate collection of words or phrases. A very faithful digitization of the human voice can legitimately occupy much less storage space than a high-fidelity musical recording of the same duration, even without the extra processing provided by the ACE Toolkit. Still, a vocabulary of just 300 words would require nearly 1.5 megabytes, assuming that the average duration of a word were one-half second, and that the publisher were satisfied with playback fidelity comparable to that of a good AM table radio. The software and vocabulary would therefore have to be supplied on several diskettes, and the entire package would probably work best only if resident on a high-capacity mass-storage device such as a hard disk. Some customers might have computer systems that could accommodate the product, but the vast majority probably would not. The publisher would therefore face a relatively high unit manufacturing cost (due to the number of disks that must be included in the package) and a relatively small market (due to the expense of peripheral equipment—i.e., a large hard disk—required for convenient operation of the software).

Depending upon the nature of our hypothetical product and its target market, the publisher might decide to go ahead and release the multiple-disk package "as is." But the decision to go to market could be made much easier, were the number of disks in the package to be reduced by simply compressing the audio data to fit in less space. If the product were a game intended for a mass audience, of course, it would not be feasible to provide the user with an extensive—and huge—library of digitized speech or sound effects. In the case of a game, not only the program, but any system software it needs, and all required data files must typically fit on one disk or at most two—a maximum storage capacity of 1.6 megabytes if standard, double-sided 3.5" disks are used. The ability to compress digitized audio to a fraction of its original size could allow the program's authors to place the entire 1.5 megabytes of sound data on one disk, exclusive of programs or other data, thus making it possible to ship the product on no more than two disks. For sound-oriented products that absolutely must be shipped on a single disk, audio data compression techniques would at least permit the developer to provide the user with a larger and more varied sound library than would be possible using uncompressed digitized audio.

How Does ACE Work?

ACE can incorporate many different compression methods; you specify which to use in making the compression and expansion calls. In the ERS, we will examine the first method to be implemented.

To prepare compressed versions of digitized audio sequences, ACE uses a technique called Adaptive Differential Pulse Code Modulation, or ADPCM. Briefly, ADPCM achieves compression by encoding not the actual audio samples themselves, but the *difference* between each sample and the value that the ADPCM algorithm *expects* that sample to have. The expected value for any given sample depends upon several factors, including previous values in the sequence, running measurements kept by the ADPCM routine concerning the accuracy of its predictions, etc. These factors constitute the model of audio wave behavior upon which the ADPCM routine bases its predictions.

Certain assumptions about audio waves are embodied within any ADPCM model, the most important being that audio waves are *continuous* and relatively *smooth*. This implies that any particular digital sample in a sequence will have a value that is reasonably close to those of its immediate predecessor and successor. So long as that fundamental assumption holds true, the ADPCM routine can track an audio wave with great accuracy. But the routine can err if the values of successive samples vary widely—as will be the case, for instance, whenever the digitized signal contains dropouts or noise spikes, or whenever the signal carries a large amount of high frequency information, relative to the nominal sampling rate. Note that typical computer data files—such as those produced, say, by a spreadsheet, a word processor program, or an assembler—tend to include data streams that would seem neither smooth nor continuous, if successive values were plotted as points on a curve. Consequently, ADPCM methods should *not* be used to compress an arbitrary data or program file, since the reconstituted file will almost certainly *not* contain exactly the same data as the original. Due to the *nature* of the errors made by an ADPCM system, however, the discrepancies between the original data and the restored data—which would be totally unacceptable in business or scientific calculations—are not only acceptable, but often even *inaudible*, to the human ear. To see why, one must examine the heart of ADPCM encoding in somewhat greater detail.

In order for ACE's ADPCM compression routine to successfully translate eight-bit data samples into compact codes of four-bits or less in width, the routine's model of wave behavior must be sufficiently accurate to ensure that the difference between the actual and

the predicted values of any particular sample can be represented by a quantity that is, at most, four-bits wide. Thus, to successfully compress a sample to four bits, the actual value must not differ from the predicted value by more than seven units in either the negative or positive directions. To compress to three bits successfully, the sample value can differ from the prediction by no more than three units in either direction. A prediction that lies too far away from the actual sample value will lead the ADPCM method to produce a compression code that represents a value different from that of the original sample. After it is reconstituted by the expansion routine, an erroneous sample value is perceived as signal distortion, i.e., as noise.

As you might expect, a good ADPCM model will introduce an imperceptible—or at worst, a barely perceptible—amount of noise into the restored audio signal. You might further expect a bad model to cause severe and discordant distortion in the restored signal, but such is not the case. Instead, less accurate models simply introduce a greater amount of "pink noise"—i.e., *static* and *hiss*—into the reconstituted signal. Interestingly enough, the perceived quality or fidelity of the restored audio signal seems unchanged to the ear; the signal, as perceived, is simply covered by a "noise blanket" that becomes thicker and more obtrusive with the declining accuracy of the ADPCM model. Stated another way, the ADPCM method manages to track the gross signal waveform rather closely, despite making even a large number of inaccurate predictions during the encoding or decoding processes. This is because of the A—for Adaptive—in ADPCM. The compression and expansion algorithms monitor the quality of their predictions, and modify their waveform behavior models accordingly on a *dynamic* basis, in order to bring the models more closely in line with the *local* behavior of the waveform being processed.

Apple's version of ADPCM incorporates a first-order predictor, and a model of audio wave behavior that is accurate enough to permit the compressed version of an eight-bit audio sample to occupy either three or four bits, as desired by the user. Because the margin of acceptable prediction error shrinks with the decreasing width of compressed samples, the likelihood of inaccurate encoding and decoding increases along with the density of the encoded data. Inaccurate encoding or decoding puts noise into the data, and so you are likely to hear more noise when auditioning a reconstituted signal that has undergone three-bit compression, than when listening to a copy of the same signal after four-bit compression. To help mitigate this problem, ACE can use offer models of wave behavior, each model optimized for either three-bit or four-bit compression, and each assuming that the original audio signal was digitized at or reasonably near a particular sample rate. You may use any one of the different wave models available through ACE by specifying the appropriate method code.

Design Motivation

The initial version of ACE uses straight ADPCM instead of other methods of data compression and expansion for the following reasons:

- *Simple implementation*. Although very sophisticated in concept, the ADPCM core routines are small and easy to implement, even for a target machine that offers a small word size and limited arithmetic support. Realization of the method in 6502, 65816, or 68000 machine language is a relatively straightforward proposition. Implementation in higher-level languages, such as Modula-2, Pascal, or C, is even easier, as was proven during development of the prototype upon which the current toolkit is founded. Other compression methods, such as Huffman encoding, must generally be implemented as longer and more complex subroutines than the ones used

by ACE. We believe that the brevity and computational simplicity of ACE's ADPCM routines will ease the task of maintaining and enhancing the ACE Toolkit.

- *Quick execution.* ADPCM compression and expansion are fast processes, partially because the processing routines themselves are short and make use of only the most rudimentary arithmetic and logical operations, and partially because any particular digitized signal is converted completely in just one pass, without recourse to backtrack, lookahead, or statistical analysis. Depending upon the efficiency of the implementation and the speed of the host processor, ADPCM processing can be done in real time or better. Some other compression methods require multiple-pass processing, and schemes based on Huffman techniques in particular cannot achieve optimum compression without engaging in an exhaustive, time-consuming statistical analysis of the *entire* sample *prior* to the encoding pass.

- *Guaranteed compression ratios.* The ACE user can be certain that any digital audio sample will be compressed to a particular fraction of its original signal size, either 1/2 or 3/8, as specified by a parameter of the compression and expansion routines. Certain other compression strategies, such as run-length encoding and tokenization, cannot guarantee compression at all. Indeed, when presented with "worst case data," eccentric implementations of those techniques can actually produce "compressed" samples that occupy *more* storage space than the originals.

We feel that the advantages of ADPCM overshadow its two major weaknesses:

- *Capacity for Error.* Most compression methods used in computerized data processing guarantee that the expanded versions of compressed data are completely identical with the original data. ADPCM compression, on the other hand, typically produces compressed data, from which the precise values of the original data *cannot* be recovered with any degree of certainty! Fortunately, the human ear tends to perceive the reconstituted sample, errors and all, as a remarkably good copy of the original.

  We believe that the nature and amount of noise impressed upon an audio signal by the ADPCM compression and expansion process is acceptable, and is in keeping with the needs and expectations of professional software developers and serious amateur programmers, who want to enhance their programs with sound effects, music, or voice. It is not appropriate or intended for use in fields where high signal fidelity is crucial, such as in professional recording, broadcasting, or serious filmmaking. As a good rule of thumb, four-bit compression is suitable in applications where the hiss present in a good stereo FM broadcast signal is acceptable, three-bit compression where the noise evident in the playback of a regular tape cassette on a reasonably affordable home deck, without noise reduction, is sufficient.

- *Effective Maximum Compression Ratio of 8-to-3 (2.67-to-1).* Using ADPCM techniques alone, it seems impractical to achieve compression ratios beyond 8-to-3, due to the extremely low margin for prediction error afforded by compression codes of less than three-bits. Although it may be possible to combine ADPCM with several other compression methods, in order to realize even greater compression than that possible using ADPCM alone, the resulting hybrid compression software would almost certainly be larger, more complex, and slower than the core routines of the ACE Toolkit. For the present design, we have opted for speed, simplicity, and guaranteed compression behavior. We leave to future toolsmiths the task of shaping hybrid compression schemes into efficient, robust toolkits.

Reference

Cummiskey, P., Jayant, S., and J.L. Flannagan, "Adaptive Quantization in Differential PCM Coding of Speech," *The Bell System Technical Journal,* September, 1973 (52:7), 1105-1118.

ACE Tool Calls

Following is a list of the calls provided by the ACE toolkit. Underlined call names denote calls that may be made by an application prior to the ACEStartup call. Whenever a status code of ACENoError is posted in the accumulator upon function return, the CPU Carry flag is *clear*. Whenever any other ACE status code is posted, an error has occurred, and the CPU Carry flag is *set*. In either case, the stack contains all *output* parameters listed for the particular call in question, if any, although their values are undefined in the presence of error conditions. It is the application's responsibility to clear the stack of any output parameters upon successful or unsuccessful function return.

About the format. Tool call descriptions are organized to promote efficient debugging of programs that use ACE. The mnemonic call name, hexadecimal call number, and parameter list are given first, as a quick guide to the proper invocation of the routine. Next, a list of return status codes is provided, along with the assumptions you may make about the conditions of ACE or the Apple IIGS system whenever each code is returned. The ACENoError code is always first in the list; the accompanying text describes the relevant state of the world after a successful call, and so, by implication, the call's normal behavior. Miscellaneous notes, if necessary, round out the tool call description, providing additional detail about the operational behavior and limits of the call, or tips for efficient programming using ACE.

ACEBootInit **Call Number $01**

Parameters: None

Return Status:

ACENoError

This call should never be made by an application. Any initialization of the ACE subsystem that may be necessary at bootstrap time has been performed.

ACEStartup **Call Number $02**

Parameters:

input	ZeroPageLoc	WORD

Return Status:

ACENoError

Any initialization of the ACE subsystem that may be appropriate at application startup has been performed. In particular, ACE will now use a region of bank zero for its direct page, starting at the address specified by ZeroPageLoc (a sixteen-bit value, since the most significant byte of the 24-bit address is implicitly zero). At the time of this writing, ACE's direct page region contains exactly one page (256 bytes). If you do not know the exact amount of contiguous bank zero space that is needed by any particular version of ACE, your program may determine the proper value by calling ACEInfo, described below. Note that ACE's direct page space should always begin on a page boundary.

ACEIsActive

The call just made was redundant; ACE has already been initialized for the current application, and already "owns" Direct Page space.

ACEBadDP

An improper starting location was given for ACE's Direct Page Area. Typically, ZeroPageLoc contained zero upon function entry, but other invalid values may also cause this error condition.

ACEShutdown **Call Number $03**

Parameters: None

Return Status:

`ACENoError`

Any cleanup of the ACE subsystem that may be appropriate at application shutdown has been performed. An application should always call this routine at the point during execution beyond which it will no longer need to make calls to ACE (prior to application shutdown in any case). Note that it is always the responsibility of the application to allocate and *deallocate* the Direct Page space used by ACE.

`ACENotActive`

The call just made was invalid; ACE cannot be shut down if it has never been started.

ACEVersion **Call Number $04**

Parameters:

output	VersionInfo	WORD

Return Status:

ACENoError

VersionInfo contains the identification number of the currently installed version of ACE, expressed in the format prescribed by the Toolbox protocol: Bit 15 is 0 if a release version of ACE is installed, and 1 if a prototype version is installed; bits 8-14 indicate the major version number; bits 0-7 contain the minor version number. For example, prototype version number 0.3 would be returned as $8003, official version number 0.3 would be returned as $0003, and official release version number 1.2 would be returned as $0102.

ACEReset **Call Number $05**

Parameters: None

Return Status:

ACENoError

This call should never be made by an application. Any initialization of the ACE subsystem that may be appropriate at machine reset (as opposed to machine bootstrap) has been performed.

ACEStatus **Call Number $06**

Parameters:

output	`ActiveFlag`	WORD

Return Status:

`ACENoError`

`ActiveFlag` contains `TRUE` (nonzero) if ACE is active or `FALSE` (zero) otherwise.

ACEInfo **Call Number $07**

Parameters:

output	InfoItemValue	LONG
input	InfoItemCode	WORD

Return Status:

ACENoError

InfoItemValue contains the ACE internal parameter value that was specified by InfoItemCode. The possible values for InfoItemCode and the corresponding parameter values returned are given in the following table:

InfoItemCode	InfoItemValue Returned by ACEInfo
0	Size, in bytes, of Direct Page space needed by ACE (high word of InfoItemValue is always zero)

ACENoSuchParam

InfoItemCode is out of range. No internal parameter corresponds to it.

(reserved) **Call Number $08**

Return Status:

`ACENotImplemented`

This call should never be made by an application. Call number $08 is reserved for future system enhancements.

ACECompress **Call Number $09**

Parameters:

input

Src	HANDLE
SrcOffset	LONG (UNSIGNED)
Dest	HANDLE
DestOffset	LONG (UNSIGNED)
NBlks	WORD
Method	WORD

Return Status:

ACENoError

ACE has successfully compressed NBlks of contiguous 512-byte blocks, containing eight-bit digital sound data, and starting SrcOffset bytes beyond the location specified by the handle Src. Data have been compressed according to the requirements of Method. The compressed version starts DestOffset bytes beyond the address specified by the handle Dest, and, for Methods 1 and 2, is (NBlks * 64 * (5-Method)) bytes in size.

Method Code	Compression method used by ACE
1	ADPCM from 8 bits to 4
2	ADPCM from 8 bits to 3

ACEBadMethod

Method contains a value that does not correspond to any compression method known to ACE.

ACEBadSrc

Src is an invalid handle.

ACEBadDest

Dest is an invalid handle.

ACEDataOverlap

With respect to NBlks, the two regions of RAM specified by Src and SrcOffset and Dest and DestOffset overlap, without being identical.

Notes:

- Because ACECompress is guaranteed to encode every original data byte using fewer than eight-bits, data sequences of arbitrary size may be compressed in place; in other words, the pairs Src and SrcOffset, and Dest and DestOffset may specify the same location in RAM.

- **IMPORTANT:** To minimize the amount of noise in the reconstituted signal, your *original* signal should be as free from noise as possible. For best results, editing, equalization, and special effects processing—such as reverb, audio compression and limiting, "time-squeezing," etc.—should be applied to the *original* signal, *before* compression, instead of to the restored signal, before playback. Finally, ADPCM compression should be the *last* operation performed upon an audio sample before it is placed on a product master disk.

ACEExpand **Call Number $0A**

Parameters:

input

Src	HANDLE
SrcOffset	LONG (UNSIGNED)
Dest	HANDLE
DestOffset	LONG (UNSIGNED)
NBlks	WORD
Method	WORD

Return Status:

ACENoError

ACE has made a reconstituted copy of previously compressed digital sound data. The compressed data sequence starts SrcOffset bytes beyond the location specified by the handle Src, and, for Methods 1 and 2, is (NBlks * 64 * (5-Method)) bytes in size. NBlks of contiguous 512-byte blocks have been produced. The restored sound sequence starts DestOffset bytes beyond the location specified by the handle Dest. Data have been expanded according to the requirements of Method.

Method Code	Expansion method used by ACE
1	ADPCM from 4 bits to 8
2	ADPCM from 3 bits to 8

ACEBadMethod

Method contains a value that does not correspond to any compression method known to ACE.

ACEBadSrc

Src is an invalid handle.

ACEBadDest

Dest is an invalid handle.

ACEDataOverlap

With respect to NBlks, the two regions of RAM specified by Src and SrcOffset and Dest and DestOffset overlap, partially or completely.

Notes:

- Because ACEExpand is guaranteed to expand every compressed sample to an eight-bit value, compressed data may *not* be reconstituted in place; in other words, the pairs Src and SrcOffset, and Dest and DestOffset may *not* specify identical or overlapping regions in RAM.

ACECompBegin **Call Number $0B**

Parameters: None

Return Status:

ACENoError

ACE is ready to compress a new audio sequence. ACE normally preserves the relevant state information from the compression process so that each successive call to ACECompress can pick up where its predecessor left off. This makes it possible to compress a large audio sequence by first dividing it into logical subsequences, then using ACECompress on each subsequence in turn. Because ACECompBegin erases the state information from any previous compression call, it should always be called before compressing the first, and only the first, of a series of subsequences. You should also call ACECompBegin before attempting to compress any audio sequence that is complete in itself (i.e., not part of a larger sequence).

ACENotActive

The call just made was invalid; ACE cannot be used if it has never been started.

ACEExpBegin **Call Number $0C**

Parameters: None

Return Status:

`ACENoError`

ACE is ready to expand a new audio sequence. ACE normally preserves the relevant state information from the expansion process so that each successive call to `ACEExpand` can pick up where its predecessor left off. This makes it possible to expand a large audio sequence by first dividing it into logical subsequences, then using `ACEExpand` on each subsequence in turn. Because `ACEExpBegin` erases the state information from any previous expansion call, it should always be called before expanding the first, and only the first, of a series of subsequences. You should also call `ACEExpBegin` before attempting to expand any audio sequence that is complete in itself (i.e., not part of a larger sequence).

`ACENotActive`

The call just made was invalid; ACE cannot be used if it has never been started.

Appendix: ACE Return Status Codes

```
$0000   ACENoError
$1D01   ACEIsActive
$1D02   ACEBadDP
$1D03   ACENotActive
$1D04   ACENoSuchParam
$1D05   ACEBadMethod
$1D06   ACEBadSrc
$1D07   ACEBadDest
$1D08   ACEDataOverlap
$1DFF   ACENotImplemented
```

MIDI Tool Set ERS

Revision 1.1.4
June 17, 1988

1.0 Introduction

The MIDI Tool Set provides a software interface that allows application developers to communicate with external musical synthesizers and other equipment that accepts the MIDI (*M*usical *I*nstrument *D*igital *I*nterface) communication protocol. The MIDI Tool Set offers the following advantages:

- **Hardware independence.** The MIDI Tool Set uses a separately-loaded device driver in order to communicate with a particular hardware MIDI interface. Application programs that use the MIDI Tool Set will be able to run on systems with different hardware interfaces.

- **Interrupt-driven operation.** The MIDI Tool Set allows input and output of MIDI data to be accomplished in the background while the CPU works on other tasks. The application programmer is not required to write his own interrupt routines.

- **Accurate clocking.** If needed, the MIDI Tool Set implements a high-speed clock using one of the generators in the DOC sound chip (leaving 13 for general use) and the first 256 bytes of DOC RAM. The clock provides very accurate time-stamping of MIDI input and output data. The resolution of the clock is 76 microseconds, providing ample resolution to minimize quantization errors when receiving large chords. The MIDI Tool Set releases the generator and the page of DOC RAM whenever the clock is not being used.

- **Fast response.** Although MIDI data may be received at very high speeds (one byte every 320 microseconds), the MIDI Tool Set allows the Apple //GS to receive this data without lossage as long as interrupts are never disabled for more than 300 microseconds. If interrupts must be disabled for longer periods of time (in an interrupt handler, for example), the MIDI Tool Set provides a simple polling mechanism which can be executed periodically in order to prevent lossage of MIDI input data.

- **Multiple formats.** The MIDI Tool Set handles MIDI input and output data in several formats. In *raw mode,* no interpretation of MIDI data is performed. Input bytes are returned to the application just as they were received. In *packet mode,* input bytes are gathered into packets, and missing running status bytes are added. The length of the packet and a time stamp are included. In *standard mode,* packets of bytes are returned in the Standard MIDI File format.

- **Error checking.** The MIDI Tool Set detects and reports a variety of error conditions, including receipt of a MIDI packet with an incorrect number of data bytes.

- **Real-time and batch operation.** The MIDI Tool Set supports two different buffering strategies that allow real-time processing of individual MIDI packets or recording and playback of larger MIDI sequences as a background process.

- **Real-time commands.** If desired, the MIDI Tool Set will report receipt of any MIDI *real-time commands* to the application immediately for appropriate action.

- **Intelligent note-off commands.** The MIDI Tool Set includes functions which turn off only those notes that it has turned on, in a specific MIDI channel or in all MIDI channels. The MIDI Tool Set can also generate note off commands which turn all possible notes off in a specific channel or in all MIDI channels.

Figure 1.1 illustrates some of the relationships between an application program, the MIDI Tool Set, a separately-loaded device driver, and the hardware MIDI interface. The device driver contains simple character input and output routines which isolate the tools as well as application programs from details of a particular hardware interface. Therefore, the MIDI Tool Set may be used with different interfaces.

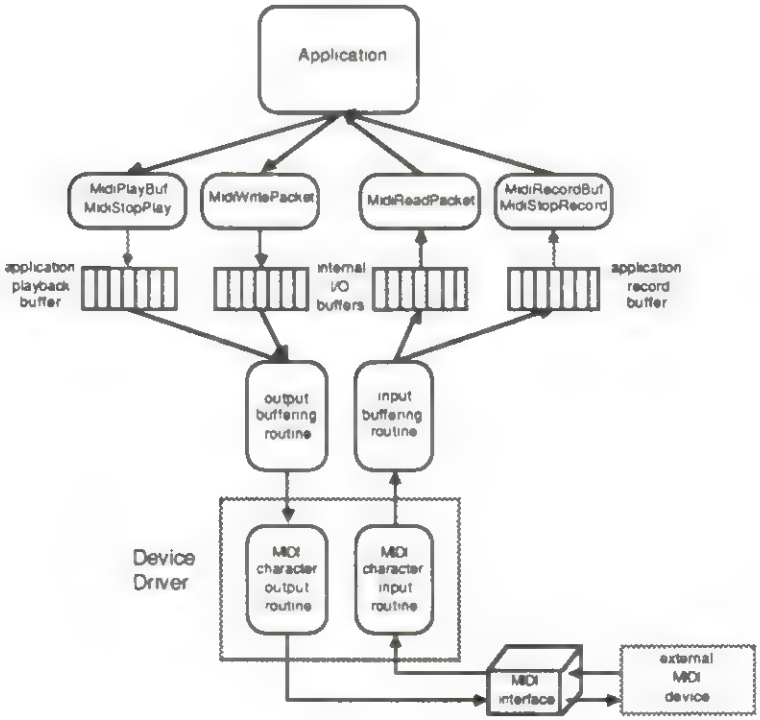


Figure 1.1

1.1 Notes on Version 1.1

This ERS explains usage and implementation of version 1.1 of the MIDI Tool Set. Although the basic functionality of these tools will probably not change greatly, many capabilities and features are still under development. A major addition planned for the future will support the Standard MIDI File format, which was adopted by the Midi Manufacturers Association in January 1988. Following is a list of changes that are planned for future releases.

MidiDevice
Implementation of the *miSelectDriver* function, which displays a device driver selection dialog. The selected device will be remembered in an initialization file so that the user only has to select his device once, and then different applications will open that device as a default. This call will be available in version 1.2 of the MIDI Tool Set.

MidiInfo
Implementation of the *miRecordAddr* and *miPlayAddr* functions to provide information on the current address being played or recorded by the *MidiPlaySeq* and *MidiRecordSeq* functions.

MidiReadPacket
Timing relationships within system exclusive packets will be noted in future versions using strategies similar to the Standard MIDI File format.

MidiWritePacket
Merge output with *MidiPlaySeq.*

MidiRecordSeq
To be implemented.

MidiStopRecord
To be implemented.

MidiPlaySeq
To be implemented.

MidiStopPlay
To be implemented.

MidiConvert
To be implemented. This function will be used to convert MIDI information between formats, and possibly to adjust time stamps for MIDI information recorded with different clock frequencies.

2.0 Tool Dependencies

The MIDI Tool Set calls the Note Synthesizer to allocate a generator in the DOC sound chip which is used to implement a high-speed clock. If the time-stamping functions of the MIDI Tool Set are not required in your application, the MIDI Tool Set may be started up without starting the Note Synthesizer first. The Note Synthesizer may be started or shutdown independently of the MIDI Tool Set whenever the MIDI Tool Set clock is stopped.

An application that uses the Note Synthesizer or the Note Sequencer to produce sounds while simultaneously receiving MIDI data must ensure that the following tool versions (or later) are used:

Sound Tools	2.3
Note Synthesizer	1.2
Note Sequencer	1.2

Previous versions of the above tools caused MIDI input data to be lost occasionally because they disabled interrupts for more than 300 microseconds. Older versions of the tools can be used if your application does not deal with MIDI input.

2.1 Using the MIDI Tool Set

After the MIDI Tool Set has been started successfully with *MidiStartUp*, it is necessary to load a device driver. This is accomplished using the *miLoadDriver* function of the *MidiDevice* call. This function requires a pointer to a parameter block containing the slot of the interface, whether the slot is internal (the printer or modem port) or an external card, and the pathname of the device driver file. In the future, the *miSelectDriver* function will enable the user to choose values for these parameters using a dialog box, and will save these selections in an initialization file so that the choices can be preserved between executions of a single program, or different MIDI programs.

MIDI device drivers are normally found in the "*/SYSTEM/DRIVERS" directory, and end with the suffix ".MIDI". Two drivers are currently available: "APPLE.MIDI" for the Apple MIDI Interface and all compatible 1Mhz-clock interfaces which plug into the modem or printer serial ports; and "CARD6850.MIDI" which supports plug-in MIDI cards based on the 6850 ACIA (such as the Passport card). A document describing device driver design for interfaces that are not compatible with either of the above will be made available if necessary.

Next, input and/or output buffers should be allocated using the *miSetInBuf* and *miSetOutBuf* functions of the *MidiControl* call. If an application

doesn't ever call *MidiReadPacket* or *MidiWritePacket*, the corresponding buffer need not be allocated.

The MIDI Tool Set is now set up to begin receiving or transmitting MIDI data. Neither input nor output will occur, however, until the processes that perform them are started. To understand this, you should be aware that all MIDI input and output operations are performed in the background. This means that an application may perform other tasks while the MIDI Tool Set is transmitting and receiving data. Since MIDI I/O can sometimes require considerable overhead, the MIDI Tool Set has been designed so that the application can start and stop these processes whenever it needs to.

When the input process is started using the *miStartInput* function of *MidiControl*, the MIDI Tool Set will begin to fill its input buffer with MIDI data, and the application should periodically retrieve the data using *MidiReadPacket* calls. When the output process is started, the MIDI Tool Set will send data in its output buffer which the application fills using calls to *MidiWritePacket*. Both input and output may be enabled simultaneously, and the Apple //GS can send and receive data at the same time.

For many applications, it is important to know exactly when a particular MIDI packet is received as well as what data the packet contains. Likewise, it is often necessary to specify exactly when a particular MIDI output packet should be sent. The MIDI Tool Set has a built-in clock in order to perform these timing operations. Because the clock uses one generator of the DOC sound chip, as well as the first 256 bytes of DOC RAM, it is normally disabled. The application may set and start the clock using *MidiClock* calls. If the clock is used, it should be started before the input process is started, because the *MidiClock* function disables interrupts for a relatively long period of time while the generator is allocated and the DOC RAM is loaded. During this time, MIDI input data could lost if the input process has already been started, and an error will be reported.

In order to accurately time MIDI events, the clock runs very fast (one tick every 76 microseconds). When the clock is running, each MIDI packet is assigned a "time stamp" which is the value of the clock at the time the first byte of the packet was received. (When the clock is stopped, each MIDI packet is assigned the static value of the clock at the time it was stopped.) On output, the MIDI Tool Set sends a MIDI packet only when the clock has reached the value of the time stamp that the application program submitted with the packet during a *MidiWritePacket* call. If the clock is stopped, no packet with a time stamp greater than the clock value will be sent.

Although it is not necessary to run the clock if you simply want to read and write MIDI packets in real time, it is sometimes beneficial to run the clock when doing output anyway. This allows the MIDI Tool Set to detect an error if the MIDI interface stops transmitting for any reason. Simply assign a time-

stamp of zero to any packet that you wish to output immediately regardless of the clock value.

At any time, the application may execute *MidiControl* calls to halt the input or output processes or *MidiClock* calls to stop the clock. This will lessen the burden on CPU cycles, and make more processing power available for other tasks to be performed by the application. The application should stop the input process any time it wishes to ignore MIDI input data. Otherwise, CPU cycles will be used to process stray MIDI commands which will simply fill up the input buffer with unwanted data. The clock releases control of its generator and the first page of DOC RAM when it is halted.

Other *MidiControl* commands allow the application to flush the input and output buffers and wait for the output buffer to empty. The *MidiInfo* call returns information about the state of the MIDI Tool Set, including the number of characters currently buffered in the input and output buffers and the current value of the clock.

In the following sections, we will give some examples of typical C code sequences using the MIDI Tool Set.

2.2 Example: Starting up the MIDI Tool Set

The *MidiStartUp* call takes two arguments: a word containing the application's memory manager id, and a word containing the address of a page-aligned three-page memory block in bank zero.

The following code demonstrates a typical startup sequence:

```
/*
 *  StartupTools()
 *
 *  Starts up the MIDI Tool Set and all of the tools it requires.
 *  NOTE:  This subroutine contains none of the error checking that
 *  would normally be performed after each tool is started.
 */

/* direct page usage */
#define DPForSound          0x0000      /* needs 1 page */
#define DPForMidi           0x0100      /* needs 3 pages */
#define TotalDP             0x0400      /* total direct page usage */

StartupTools()
{
        static struct {
                word Length;
                word [5*2];
        } ToolTable = {
                5,                                  /* number of items in list */
                1,0x0101,                           /* Tool Locator */
                2,0x0101,                           /* Memory Manager */
                8,miSTver,                          /* Sound Tools */
                25,miNSver,                         /* Note Synthesizer */
                miToolNum,0x0000                    /* MIDI Tool Set */
        };

        MiDriverInfo DriverInfo;                    /* device driver information */
        MiBufInfo InBufInfo,OutBufInfo;             /* buffer information */
        handle zphandle;                            /* zero page handle */
        ptr zpptr;                                  /* zero page pointer */


        TLStartup();                                /* Tool Locator */
        id = MMStartUp();                           /* Memory Manager */

        /* allocate direct pages for tools */
        zphandle = NewHandle((long) TotalDP,
                             (word) id,
                             (word) attrBank+attrPage+attrFixed+attrLocked,
                             (long) 0);
        zpptr = *zphandle;

        /* load RAM-based tools */
        LoadTools(&ToolTable);

        SoundStartUp((word)(zpptr + DPForSound);
        NSStartUp(0,01);                            /* choose your own update rate */
        MidiStartUp(id,(word)(zpptr + DPForMidi);
```

```
/* load device driver */
DriverInfo.slot = 2;                        /* use modem port */
DriverInfo.external = 0;                    /* internal */
strcpy(DriverInfo.file,"\p*/system/drivers/applemidi");
MidiDevice(miLoadDrvr,&DriverInfo);

/* allocate input and output buffers */
InBufInfo.size = 0;                         /* default size */
InBufInfo.address = 0;                      /* MIDI Tool Set will allocate it */
MidiControl(miSetInBuf,&InBufInfo);
OutBufInfo.size = 0;                        /* default size */
OutBufInfo.address = 0;                     /* MIDI Tool Set will allocate it */
MidiControl(miSetOutBuf,&OutBufInfo);
```

2.3 Example: Reading time-stamped MIDI data

The following subroutine demonstrates a simple method of recording time-stamped MIDI input data until a key on the computer's keyboard is depressed. Note that the *MidiRecordSeq* function will normally be used to record a sequence of MIDI notes (when it is implemented), because the overhead of many tool calls and memory moves is avoided using that call.

```
/*
 *  RecordMidi()
 *
 *  Record incoming MIDI data with time stamps until our buffer is full or
 *  the user presses a key.
 */

#define BufSize       20000                     /* space for medium-size sequence */
char Buf[BufSize];                              /* buffer to hold MIDI sequence */
int BufIndex = 0;                               /* index into buffer */

RecordMidi()
{
        int packetsize;                         /* size of packet read */
        char c;
        char *Keydata = 0xE0C000;
        char *Keystrobe = 0xE0C010;

        MidiControl(miFlushInput,01);           /* discard contents of input buf */
        MidiClock(miSetFreq,01);                /* set clock to default frequency */
        MidiClock(miSetClock,01);               /* clear the clock */
        MidiClock(miStartClock,01);             /* start the clock */
        MidiClock(miStartInput,01);             /* start MIDI input */

        BufIndex = 0;

        while ((*Keyboard & 0x80) == 0) {       /* until user hits a key */
                packetsize = MidiReadPacket(Buf+BufIndex,BufSize-BufIndex);
                if (_toolErr) {
                        if (_toolErr == miArrayErr) break;  /* our buf is full */
                        printf("MIDI error $%4.4X\n",_toolErr);

                else
                        BufIndex += packetsize;     /* update buffer index */


        c = *Keystrobe;                         /* clear keyboard */

        /* stop recording */
        MidiControl(miStopInput,01);            /* stop MIDI input */
        MidiClock(miStopClock,01);              /* stop clock */

         * print statistics */
        printf("Bytes recorded: %d\n",BufIndex);
        printf("Max. bytes buffered: %ld\n",MidiInfo(miMaxInChars));
```

2.4 Example: Output of time-stamped MIDI data

The following code segment demonstrates a simple subroutine that will continuously playback time-stamped MIDI data which has previously been recorded by the subroutine above. As above, this subroutine is provided for demonstration purposes. When the *MidiPlaySeq* function is available in future versions of the MIDI Tool Set, it will normally be used to play a MIDI sequence.

```
/*
 *  PlayMidi()
 *
 *  This routine plays back the MIDI data previously recorded in
 *  the global buffer "Buf".
 */

PlayMidi()
{
        char *Keydata = 0xE0C000;
        char *Keystrobe = 0xE0C010;
        long firsttime;
        int i;
        char c;

        if (BufIndex == 0) {
                printf("You must record some MIDI data first\n");
                return;
        }

        /* find the first time stamp in the sequence and subtract a little */
        firsttime = *((long *) (Buf+2));
        if (firsttime > 0x200) firsttime -= 0x200;
        else firsttime = 0;

        MidiControl(miFlushOutput,(long)(0xFFFF << 16)); /* empty output buf */
        MidiClock(miSetClock,firsttime);            /* set clock to 1st time stamp */
        MidiClock(miStartClock,0l);                 /* start clock */
        MidiControl(miStartOutput,0l);              /* start output */
        i = 0;

        /* play song again */
        while ((*Keyboard & 0x80) == 0) {
                i += MidiWritePacket(Buf+i);        /* write next packet */
                if (i == BufIndex) {
                        MidiControl(miWaitOutput,0l);       /* wait for end of song */
                        MidiClock(miSetClock,firsttime);    /* restart clock */
                        i = 0;
                }
        }

        c = *Keystrobe;                             /* clear keyboard */
        MidiControl(miFlushOutput,0x101);           /* turn notes off, all channels */
        MidiClock(miStopClock,0l);                  /* stop clock */
        MidiControl(miStopOutput,0l);               /* stop output */
}
```

3.0 Restrictions

In this section, we will mention various restrictions and usage hints that can help or hinder operation of the MIDI Tool Set.

The MIDI Tools are not designed to work with AppleTalk enabled. The reason for this restriction is that the Apple //GS is not fast enough to process high-speed AppleTalk interrupts and high-speed MIDI interrupts simultaneously. If an application *must* run with MIDI and AppleTalk enabled at the same time, occasional MIDI input errors should be expected, and MIDI output may be slightly delayed at times. For most programs, even one MIDI input error may be difficult to deal with, therefore software publishers should advise customers against using MIDI software with AppleTalk enabled.

For similar reasons, MIDI input may be lost during disk accesses. Tools such as the Event Manager, the Dialog Manager, and operations such as cursor updates also lock out interrupts in ways that preclude error-free receipt of MIDI data. Although means for curing these difficulties are currently being investigated, programs written with the tools provided on System Disk 3.2 will have to refrain from these operations while MIDI input is enabled.

Current versions of the Sound Tools, Note Synthesizer, and Note Sequencer have been modified in order to be "MIDI-friendly", so these tools may be fully used while MIDI input is enabled. For example, we have written simple test programs that demonstrate the Note Sequencer playing notes on the GS (and optionally producing MIDI output) while MIDI input is simultaneously being accepted and translated to Note Synthesizer commands, which are played on the GS along with the notes produced by the Note Sequencer. Such a program allows us to "play along" with sequences that are running on the GS with no additional equipment other than a MIDI keyboard and MIDI interface.

The MIDI Tool Set uses one generator on the Ensoniq DOC chip to implement its high-speed clock. While the clock is running, this generator cannot be reused or reallocated. If you use the *AllocGen* call in the Note Synthesizer to allocate generators, this restriction simply means that one of the generators will never be returned for your use as long as the MIDI Tool Set clock is active (leaving 13 generators remaining for general use). The first 256 bytes of Sound RAM are also reserved by the clock, and may not be altered until the clock is stopped. There are many applications which will not require accurate time-stamping of input data, and will always generate data to be output immediately. In these cases, use of the clock is not necessary, and the above restrictions need not concern you (also, the Note Synthesizer need not be started in this case). The MIDI Tool Set releases the generator and the first page of DOC RAM whenever the clock is stopped by the application.

These restrictions preclude usage of the Sound Tools' Free-form Synthesizer while the clock is running, because the Free-form synthesizer may overwrite the first 256 bytes of Sound RAM. In practice, the Sound Tools' Free-form Synthesizer should not be used while MIDI input is enabled, because the frequency and duration of Sound Tools interrupts will prevent MIDI input from being serviced often enough.

The Apple //GS has only enough processing power to deal reliably with MIDI input data arriving from one source. Although it is physically possible to connect more than one MIDI hardware interface to the //GS, the MIDI Tool Set supports only one. It seems unlikely that many situations will arise where usage of more than one MIDI interface is necessary.

Even with a single MIDI interface, a //GS must work hard to keep up with high-speed MIDI input data that can arrive at a rate of 320 microseconds per byte. A MIDI interface connected to the computer through one of the serial ports has a three-byte hardware buffer which, in theory, provides approximately 1 millisecond before MIDI data is lost. Card-based interfaces such as the Passport card often provide buffering of only one character, however. To maintain maximum reliability with all interfaces, an application program should refrain from disabling interrupts while reading MIDI data, or at least spend less than 300 microseconds at a time with interrupts disabled.

In some cases, compliance with the above restriction is not possible. For example, interrupt-servicing routines may not reenable interrupts because the //GS interrupt handler is not reentrant. To avoid the risk of losing MIDI input data in these situtations, a polling scheme has been implemented. When interrupts have been disabled for any length of time approaching 270 microseconds, the routine that is currently executing must call the "MidiInputPoll" vector located at $E11DD8. If the MIDI Tool Set is active, and a device driver is present, and MIDI input has been started, this routine will unload any MIDI characters that have arrived and buffer them in the MIDI Tool Set buffer just as if a normal MIDI interrupt had caused this action. If any of these conditions is not satisfied, the vector will immediately RTL to the caller. The vector may be called multiple times if necessary, and must be called at least once for every 270 microseconds spent in your code.

The polling call is simple. The contents of the A, X, and Y registers will be modified, but the direct page and data bank registers are preserved. The call must be made in full native mode (long accumulator and index registers), and returns in the same state:

```
JSL $E11DD8
```

WARNING: The above call should not be made before the MIDI Tool Set has been loaded unless you are using Sound Tools version 2.3 or later. Failure to observe these conditions will result in a system crash.

For applications which need to know, the following estimates are provided to help you estimate the cost of a MIDI polling call. An immediate RTL will cost about 3 microseconds, an unfruitful polling operation will cost about 50 microseconds, and successful polling will take about 150 microseconds per character read. Therefore, the polling routine could take as long as 450 microseconds in the event that 3 characters are buffered in the SCC serial chip, but this assumes that interrupts were locked out for about 1 millisecond prior to the polling call, and this should be avoided to maintain compatibility with other MIDI interfaces.

If you must resort to use of this polling mechanism, it is necessary to guarantee that it is called every 270 microseconds. For marketed software products, it is a good idea to test your solution with a full-speed source of MIDI input and a MIDI interface that does minimal buffering of MIDI data (such as the Passport card) in order to ensure compatibility with all MIDI interfaces.

The *MidiReadPacket* and *MidiWritePacket* calls perform polling whenever they are called with interrupts disabled. This means that it is permissable to call these routines from within interrupt-servicing routines (the Note Synthesizer user interrupt vector, for example, or the notification routines specified in the *miStartInput* and *miStartOutput* functions of *MidiControl*). Because other Apple //GS tool sets do not perform such polling, other tool calls should be avoided inside interrupt-servicing routines if MIDI input has been started.

WARNING: Only *MidiReadPacket* and *MidiWritePacket* should be called from within an interrupt-servicing routine. Other MIDI Tool Set calls are not reentrant, and could eventually cause the system to crash if called from within an interrupt.

As long as MIDI data is processed in a timely fashion (either through MIDI interrupts or the polling mechanism), an application program can afford to be a little more leisurely about retrieving the MIDI packets from the MIDI Tool Set. The difference between the speed at which MIDI data is received and the speed at which an application program can process the packets (considering that a significant percentage of CPU cycles are spent just buffering the incoming data) determines the size of the input buffer. The application determines the size of the input buffer using a call to the *MidiControl* function. Data will be lost if the input buffer is too small and it fills up before the application can empty it. By increasing the size of the input buffer, data lossage due to buffer overflow will be reduced. Competing interrupt-driven processes such as the Note Synthesizer with a non-zero update rate or the Note Sequencer will reduce the amount of processor time available for processing MIDI data, and will therefore require increased buffer sizes.

The size of the input buffer may also be determined by the size of the largest MIDI system-exclusive packet you intend to receive. Currently, the default size of the input and output buffers is 8K bytes. This default size was chosen because it is large enough to hold two very large system-exclusive packets. The input buffer should be at least twice as big as the largest MIDI packet you think the program will receive, because the MIDI Tool Set does not return a MIDI packet to the application program until it receives the entire packet (when it is used in *packet mode*). During the time *MidiReadPacket* is copying a large packet from the input buffer into the application's array, enough room should be left in the input buffer to accomodate the simultaneous receipt of a second large packet.

Once your application is running, it is a good idea to gather some statistics on the maximum number of characters in the input buffer at any time. This can be accomplished using calls to the *MidiInfo* routine. If the maximum number of characters in the input buffer exceeds half of the buffer size during normal usage, it may be necessary to allocate a larger input buffer, speed up the rate at which MIDI packets are removed from it, or reduce the overhead of competing interrupt-driven processes. The maximum buffer size is 32767 bytes.

In many time-critical situations, it is desirable to call routines in the MIDI Tool Set directly without incurring the overhead of a general tool call. The following C code demonstrates a method for doing this. This code saves approximately 85 microseconds per call, which can quickly add up when an application wishes to perform many *MidiReadPacket* or *MidiWritePacket* calls.

```
word MidiDP2;                          /* the address of the second direct page
                                          allocated for the MIDI Tool Set */

ptr MidiReadAddr;
pascal int MidiReadGlue();

main()
{
                .
                .
                .
        MidiStartUp(id,(word)zpptr);
        MidiDP2 = zpptr + 0x100;       /* IMPORTANT: must point to second direct page */

        MidiReadAddr = GetFuncPtr(0,0x0D20); /* find addr. of MidiReadPacket */
                .
                .
                .

        if (MidiReadGlue(Buf,BufSize) && !_toolErr) {
                /* do something with packet in Buf */
        }
}


asm(MidiReadGlue)
{
        jsl MidiReadGlue2
        sta _toolErr
        rtl
}

asm(MidiReadGlue2)
{
        lda MidiReadAddr+1
        pha
        phb
        lda MidiReadAddr
        sta 1,s
        lda MidiDP2
        rtl
```

The following code demonstrates a similar tool call acceleration technique in assembly language:

```
;
;  look up address of MidiWritePacket (for example)
;
                    pushlong #0                 ; space for result
                    pushword #0                 ; system tool
                    pushword #$0E20             ; tool and function number
                    _GetFuncPtr
                    pla                         ; save address
                    sta MidiWriteAddr
                    pla
                    sta MidiWriteAddr+2

                     .
                     .
                     .

;
;  do this instead of _MidiWritePacket
;
                    jsl MidiWriteGlue

                     .
                     .
                     .

;
;  IMPORTANT NOTE:  The variable "MidiDP2" contains the address of
;  the SECOND direct page of the three zero pages allocated for
;  the MIDI Tool Set when MidiStartUp is called.  If MidiStartup
;  is given a starting address of X, then MidiDP2 = X + $100.
;
MidiWriteGlue       phk                         ; push extra RTL address
                    lda #GlueReturn-1           ;   to simulate Tool Locator
                    pha
                    lda MidiWriteAddr+1         ; simulate call
                    pha                         ;   to MidiWritePacket
                    phb
                    lda MidiWriteAddr
                    sta 1,s
                    lda MidiDP2                 ; accumulator must contain the
GlueReturn          rtl                         ;   MIDI Tool Set direct page

MidiWriteAddr       ds 4
```

4.0 Time stamp Clock

In this section, we give details on the implementation of the MIDI time stamping clock. Please be aware that this technique will not be used on future Apple // machines, when hardware timers may be available. If an application developer uses a similar technique to implement timing, such solutions will not be compatible with future machines, in which different implementations of the sound circuitry are also possible.

In order to properly time stamp MIDI input data as it is received, it is necessary to have a high-speed clock. The resolution of this counter must be less than a millisecond. When a long stream of MIDI data is received during a short interval (such as when the user plays a large chord on a MIDI keyboard), it is necessary to accurately time stamp each note. If the clock resolution is insufficient, the notes of a large chord may be displaced slightly in time when they are played back, causing "flams".

On the Apple //GS, interrupts at a rate of 1000 per second (or more) are not only impossible to generate, but would overwhelm the processor if it were simultaneously attempting to process large amounts of MIDI data.

To solve this problem, we use a generator on the Ensoniq digital oscillator chip (DOC) to implement a high-speed counter. This method employs a 256-byte waveform filled with consecutive integers (range $01 to $FF). The DOC is set to "play" this waveform with zero volume. When a MIDI character is received, the time stamping routine reads the value of the byte that the DOC is currently "playing" from this waveform. The higher-order bytes of the current time stamp are incremented by interrupts that are generated each time the Ensoniq generator completes one pass through the waveform (once every 19.45 milliseconds).

During each interrupt that is generated when the Ensoniq chip reaches the end of the waveform, the upper three bytes of the four-byte clock are incremented. If the upper three bytes are greater than the corresponding bytes of the time stamp at the head of the output buffer, the packet at the head of the buffer is output immediately. If the upper three bytes of the clock and the time stamp are equal, the processor polls the Ensoniq chip until the waveform byte is greater than or equal to the low byte of the time stamp. During this polling period, interrupts may be disabled for as long as 1/50 of a second. To prevent lossage of MIDI input data during this interval, the device driver is called upon to poll the receiver and empty any arriving MIDI data.

MidiBootInit **function $0120**

Inputs:
none

Outputs:
none

Errors:
none

This call is made when the MIDI Tool Set is loaded.

This function should not be called by an application program.

MidiStartUp **function $0220**

Inputs:
Word - User ID for memory manager
Word - Address of page-aligned, 3-page block in bank zero

Outputs:
none

Errors:
none

The MIDI Tools startup call is called by an application that wishes to use the MIDI Tool Set. This call must be the first call made by an application to the MIDI Tool Set.

In most circumstances, the application must subsequently call *MidiDevice* in order to load a device driver and *MidiControl* to allocate an input and/or output buffer before normal usage of the MIDI Tool Set can take place.

MidiShutDown **function $0320**

Inputs:
none

Outputs:
none

Errors:
none

The application calls this function when it is done using MIDI. *MidiShutDown* deallocates the input and output buffers, stops the time stamping clock and deallocates the generator used for time stamping, and calls the driver routine to shut down the hardware interface. The device driver is then unloaded from memory. This activity takes place immediately (any buffered MIDI data is discarded). The application should therefore take steps to ensure that all recent MIDI output has actually been sent out (see *MidiControl* function).

MidiVersion **function $0420**

Inputs:
Word - space for version number

Outputs:
Word - version number

Errors:
none

This call returns the MIDI Tools version number. The format of this number is specified in the Tool Locator documentation.

MidiReset **function $0520**

Inputs:
none

Outputs:
none

Errors:
none

This function shuts down the MIDI Tool Set under the assumption that a system reset occurred. Most of the same tasks are performed as in *MidiShutDown*, but shutting down the hardware interface might require different assumptions in this case than shutting it down normally, so *MidiReset* calls the driver's reset routine instead of the driver's shutdown routine.

MidiStatus **function $0620**

Inputs:
Word - space for result

Outputs:
Word - boolean value

Errors:
none

This function returns one word on the stack. The returned value is $FFFF if the MIDI Tool Set is active, and zero if it's not.

MidiControl **function $0920**

Inputs:
Word - number of function to perform (see below)
Long - argument used by various functions

Outputs:
none

Errors:
see individual functions below

MidiControl contains many individual functions which control the operation of the MIDI Tool Set. A particular control function is selected by the value of a word parameter passed on the stack. *MidiControl* also requires a long parameter, which is used by some of the control functions and ignored by others.

Control function numbers are as follows:

0 (miSetRTVec) Set real-time vector. The long parameter contains the address of a service routine in the application program that the MIDI Tool Set calls when MIDI real-time commands are received. A value of zero disables the service routine. The service routine must not enable interrupts. If it executes for longer than 300 microseconds, the routine must call the MIDI polling vector at least once every 300 microseconds (see section 3.0 of this ERS) to avoid data lossage. The service routine may call *MidiReadPacket* or *MidiWritePacket,* but no other MIDI Tool Set routines. The real-time MIDI data is passed to the indicated routine in the low byte of a word on the stack (above the RTL address). This word must be left on the stack (note that this follows C calling conventions, not Pascal calling conventions). When the service routine is called, the data bank register is set to the value it had when the *MidiStartUp* call was made, but the direct page register points to one of the MIDI Tool Set's direct pages and must be preserved.

1 (miSetErrVec) Set real-time error vector. The long parameter contains the address of a service routine in the application program that the MIDI Tool Set calls when a real-time error condition occurs. A value of zero disables the service routine. The service routine must not enable interrupts. If it executes for longer than 300 microseconds, the routine must call the MIDI polling vector at least once every 300 microseconds (see section 3.0 of thie ERS) if MIDI data is being received. The service routine may call *MidiReadPacket* or *MidiWritePacket,* but no other MIDI Tool Set routines. The error is passed in a word on the stack (above the RTL address). This word must be left on the stack (note that this follows C calling conventions, not Pascal calling conventions). When the service routine is called, the data bank register is set to the value it had when the *MidiStartUp* call was made, but the direct

page register points to one of the MIDI Tool Set's direct pages and must be preserved.

Possible error codes passed to the error-servicing routine are:

miClockErr ($200A): The time stamp clock value just wrapped. The period of the clock is 45 hours, so this is an unlikely error, but potentially disastrous.

miDevNoConnect ($2084): If the time stamp clock is active, this error will be generated approximately once every ten seconds if output becomes blocked for a device-related reason.

2. (miSetInBuf) Allocate input buffer. The long parameter contains a pointer to a 6-byte record which contains a word indicating the size of the input buffer, followed by a long indicating the address of the input buffer. If the address is zero, the MIDI Tool Set will allocate the input buffer itself. If the size is also zero, the MIDI Tool Set will allocate a default buffer size (8K bytes). If the application has already allocated a buffer, it must be non-purgeable, fixed location, and must not cross bank boundaries.

Possible errors:

miArrayErr ($2002): buffer size must be 32 <= *size* <= 32767.

$02xx: memory could not be allocated from the Memory Manager.

3 (miSetOutBuf) Allocate output buffer. The long parameter contains a pointer to a 6-byte record which contains a word indicating the size of the output buffer, followed by a long indicating the address of the output buffer. If the address is zero, the MIDI Tool Set will allocate the output buffer itself. If the size is also zero, the MIDI Tool Set will allocate a default buffer size (8K bytes). If the application has already allocated a buffer, it must be non-purgeable, fixed location, and must not cross bank boundaries.

Possible errors:

miArrayErr ($2002): buffer size must be 32 <= *size* <= 32767.

$02xx: memory could not be allocated from the Memory Manager.

4 (miStartInput) Start MIDI input. This routine initiates an interrupt-driven process which reads MIDI data into the MIDI Tools' input buffer. This data may be retrieved by the application using *MidiReadPacket* calls.

The long parameter contains the address of a service routine that is called whenever the first packet is available in a previously empty input buffer (zero disables the service routine). The service routine must not enable interrupts. If it executes longer than 300 microseconds, it must call the MIDI polling vector (see section 3.0 of this ERS) to avoid input data loss age. The service routine may call *MidiReadPacket* or *MidiWritePacket*, but cannot call any other MIDI Tool Set routines. When the service routine is called, the data bank register is set to the value it had when the *MidiStartUp* call was made, but the direct page register points to one of the MIDI Tool Set's direct pages and must be preserved. The service routine will be called immediately if a complete packet is already available in the input buffer at the time the *miStartInput* function is called.

Any data in mid-transmission at the time input is started will be discarded until the first MIDI status byte is received.

Possible errors:

miNoBufErr ($2007): an input buffer must be allocated using the *miSetInBuf* option before the input process is started.

miNoDevErr ($200C): a device driver must be loaded using the *miLoadDrvr* function of *MidiDevice*.

miConflictErr ($200B): the function *MidiRecordSeq* is currently consuming all MIDI input and placing it in an external buffer provided by the application. The *miStartInput* function must be reexecuted when the *MidiRecordSeq* process has been terminated.

5 (miStartOutput) Start MIDI output. This routine initiates an interrupt-driven process which writes MIDI data from the MIDI Tools' output buffer. The application places data in the output buffer using *MidiWritePacket* calls. The output process may be active at the same time the *MidiPlaySeq* process is writing MIDI data from an external buffer provided by the application. In this case, both output sources are merged into one time-sorted output stream.

The long parameter contains the address of a service routine that is called whenever the output buffer has been completely emptied (zero disables the service routine). The service routine must not enable interrupts. If it executes for longer than 300 microseconds, the service routine must call the MIDI polling vector (see section 3.0 of this ERS) if MIDI input has been started. The routine may call *MidiReadPacket* or *MidiWritePacket*, but no other MIDI Tool Set routines. When the service routine is called, the data bank register is set to the value it had when the *MidiStartUp* call was made, but the direct page register points to one of the MIDI Tool Set's direct pages and must be preserved. The service routine will be called immediately if the output buffer is empty at the time the *miStartIOutput* function is called.

Possible errors:

miNoBufErr ($2007): an output buffer must be allocated using the *miSetOutBuf* option before the output process is started.

miNoDevErr ($200C): a device driver must be loaded using the *miLoadDrvr* function of *MidiDevice*.

6 (miStopInput) Stop MIDI input. MIDI data is ignored until next *miStartInput* call.

Possible errors:

miConflictErr ($200B): The function *MidiRecordSeq* is currently consuming all MIDI input and placing it in an external buffer provided by the application. Use *MidiStopRecord* instead.

7 (miStopOutput) Stop MIDI output. Output from the output buffer is halted until the next *miStartOutput* call, when output will resume where it left off.

8 (miFlushInput) Flush input buffer. The current contents of the input buffer are discarded.

Possible errors:

miNoBufErr ($2007): no input buffer has been allocated.

9 (miFlushOutput) Flush output buffer. The current contents of the output buffer are discarded. The long parameter dictates the method by which this is accomplished.

Long parameter value	Action
$0000 00XX	Wait for the current packet to finish transmission, then turn off all notes that have not been turned off in channel XX ($00 through $0F). If XX = $10, turn off notes in all channels.
$0001 00XX	Wait for the current packet to finish transmission, then turn off all possible notes (pitch $00 through $7F) in channel XX ($00 through $0F). If XX = $10, turn off every possible note in all channels (this will take a few seconds).
$FFFF XXXX	Discard the contents of the input buffer immediately without turning off any notes.

The "note off" side effects of this command may be useful even if you know that there is no data in the output buffer. Note off commands are generated for notes originating from both *MidiWritePacket* and *MidiPlaySeq*. Therefore, it is usually a good idea to call this function after you halt a playing sequence prematurely using *MidiStopPlay*.

Some synthesizers may require a short delay between the high-speed note-off commands generated by this function. See the *miSetDelay* function below.

Possible errors:

miNoBufErr ($2007): no output buffer has been allocated.

miOutOffErr ($2005): output must be enabled in order to generate note off commands.

10 (miFlushPacket) Flush input packet. Discard the next input packet (if a complete packet is available). This function will ordinarily be used to discard large system exclusive packets that an application does not wish to see (and which might not fit in a small array that the application passed to contain the next packet). This is especially useful for discarding large system-exclusive packets in an application which does not need to process them.

Possible errors:
miNoBufErr ($2007): no input buffer has been allocated.

11 (miWaitOutput) Wait for output buffer to clear. Execution returns only when the MIDI Tools' output buffer has become empty. This function might not return if output is not enabled!

Possible errors:
miNoBufErr ($2007): no output buffer has been allocated.

12 (miSetInMode) Set input mode. The long parameter selects between the *raw* (parameter = 0), *packet* (parameter = 1), and *standard* (parameter = 2) input formats for *MidiRecordSeq* and *MidiReadPacket* (raw and packet modes only). In raw mode, no interpretation of input data is made, and bytes are returned exactly as they are received (maximum of 4 MIDI characters per packet). In packet mode, the MIDI Tool Set attempts to return input data to the application in complete MIDI packets. In standard mode, MIDI packets are returned in a compact representation compatible with the Standard MIDI File format.

The current contents of the input buffer are discarded whenever this call is made, because the input buffer cannot accomodate input data with differing formats simultaneously.

13 (miSetOutMode) Set output mode. This long parameter selects between the *raw* (parameter = 0), *packet* (parameter = 1), and *standard* (parameter = 2) formats for *MidiPlaySeq* and *MidiWritePacket* (raw and packet modes only). In raw mode, no attempt is made to track note on and note off commands in order to support the intelligent all notes off function of *miFlushOutput*. In packet mode, this note on and note off tracking is performed. In standard mode, packets are represented using a compact format compatible with the Standard MIDI File format (note tracking is also enabled). In all modes, running status optimization is performed during output unless disabled using the *miOutputStat* call.

When switching to raw mode, the internal array is cleared which the MIDI Tool Set uses to mark which notes are currently on (see *miClrNotePad*), since the information in this array will become outdated if note on or note off commands are issued in raw mode.

NOTE: Although raw and packet output formats may exist in the output buffer simultaneously, note tracking may not perform predictably. Therefore, it is a good idea to wait for an empty output buffer before switching modes.

14 (miClrNotePad) Clear note pad. The MIDI Tool Set records which notes are on whenever it outputs a note on or note off command. (These note commands can originate from *MidiWritePacket, MidiPlaySeq,* or both.) This function allows the application to make the MIDI Tool Set believe that all notes are currently off in the MIDI channel specified by the long parameter ($00000000 through $0000000F) or all 16 channels if the long parameter is $00000010. This may be useful if some external situation has caused the

synthesizer to turn off all notes (the synthesizer's power has been cycled, for example).

15 (miSetDelay) Set MIDI delay value. Inexpensive synthesizers are sometimes unable to handle MIDI data sent at full MIDI speeds. The *miSetDelay* function allows the application to specify a minimum delay between output packets. The delay value is specified in the low word of the long parameter in units of approximately one-quarter of a MIDI character time (76 microseconds, to be exact). A value of four, for example, would specify that a time delay equivalent to the transmission of one MIDI character must elapse between the transmission of each MIDI packet.

If the time stamp clock is running, the MIDI Tool Set can use it to implement this delay. If the time stamp clock is stopped, the MIDI Tool Set must use code loops to simulate the delay. This will increase the amount of time spent in the MIDI Tool Set. Furthermore, if MIDI input is enabled, the MIDI Tool Set must check for the arrival of input data during the delay, thus lengthening the delay unpredictably. Therefore, if MIDI data is being received, the delay mechanism will be most accurate if the clock is also started.

The delay affects all MIDI output (*MidiWritePacket*, *MidiPlaySeq*, and the *miFlushOutput* function in *MidiControl*). The default value of the delay is zero. The delay value may be set at any time. Many synthesizers may need to set a delay value before performing the "all possible notes off" function with *miFlushOutput,* due to the quantity and speed of the data that is produced.

16. (miOutputStat) Enable/disable running status output. By default, the MIDI Tool Set normally omits the status bytes of MIDI output packets whenever possible using the "running status" convention detailed in the specification of MIDI. Although this running status optimization speeds transmission and relieves some burden from the computer, it can cause a major malfunction if the MIDI Tool Set and the synthesizer ever disagree on what the current status byte is. (This can happen if the power is cycled on the synthesizer in the middle of a sequence, or a MIDI cable becomes unplugged, for example.) For maximum reliability (at the cost of performance), this call allows the application to defeat running status optimization.

The low word of the long argument determines whether running status is enabled or disabled. A value of $0000 disables running status, and a value of $0001 enables it. Regardless of the value, the next packet to be output will include a status byte. Therefore, it may be useful to call this function occasionally to enable running status, even if you know running status is already enabled. This will ensure that the MIDI Tool Set and the synthesizer agree on the current status byte.

17. (miIgnoreSysEx) Enable/disable receipt of system exclusive packets. Unless your application recognizes system exclusive packets for a particular synthesizer, it will probably wish to ignore system exclusive packets on input. This function allows the MIDI Tool Set to ignore

such packets (any packet beginning with $F0). In the disabled mode, system exclusive packets are not buffered and not returned to your application.

Receipt of system exclusive packets is disabled if the low word of the long argument is $0001, and enabled if the low word is $0000.

MidiDevice **function $0A20**

Inputs:
Word - function number (see below)
Long - pointer to device driver information

Outputs:
none

Errors:
see functions below

MidiDevice allows selection, loading, and unloading of device drivers to be used with the MIDI Tool Set. The *miSelectDrvr* and *miLoadDrvr* functions interpret the long parameter as the address of a 69-byte device driver information record which contains the following information:

Offset	Contents
0 - 1	slot number ($0000 through $0007)
2 - 3	slot external ($0001) or internal ($0000)
4 - 68	pathname of device driver file (Pascal-type string)

Function numbers are:

0 (miSelectDrvr) Display device driver selection dialog. A device driver selection dialog is presented to the user. The default selections come from the initialization file */SYSTEM/DRIVERS/MIDI.SETUP. The values will be altered in the device driver information record according to the user's choices. These choices will also be saved in the initialization file so the same choices will be available across multiple executions of an application or different applications. The resulting record is suitable for submission to the *miLoadDrvr* function.

Possible errors:
THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

1 (miLoadDrvr) Load a device driver. The device driver indicated by the device driver record is loaded into memory (after any previous drivers are shutdown and unloaded), and initialized.

Possible errors:
miDriverErr ($2008): indicated file did not have the correct file type and aux type for MIDI device driver.
miDevNotAvail ($2080): device is not available in the indicated slot.
miDevSlotBusy ($2081): indicated slot is already in use.
miDevBusy ($2082): device is busy.
miDevNoConnect ($2084): device is disconnected.

miDevVersion ($2086): device driver is incompatible with the ROM version or machine type.

miDevIntHndlr ($2087): another program has stolen the interrupt vector that the device driver needs to use.

$20xx: other device driver error.
$00xx: ProDOS 16 error.
$02xx: Memory Manager error.
$11xx: Loader error.

2. (miUnloadDrvr) Unload a device driver. Shutdown and unload the currently active device driver. This call might be made to free some memory when MIDI is no longer needed by an application program. If the application is currently receiving or transmitting MIDI data, these processes are terminated.

MidiClock **function $0B20**

Inputs:
Word - number of function to perform (see below)
Long - argument used by various functions

Outputs:
none

Errors:
see individual functions below

MidiClock contains functions which control the operation of the optional time stamp clock, which is used to provide time stamps for input data, and which initiates the output of output data at the appropriate times. The clock counts at a rate of one tick every 76 microseconds. A particular clock control function is selected by the value of a word parameter passed on the stack. *MidiClock* also requires a long parameter, which is used by some of the control functions and ignored by others.

Function numbers are as follows:

0 (miSetClock) Set time stamp clock value. The long parameter contains a value which will be the new value of the clock. This value must be positive (the most significant bit must be zero). The least significant byte of the clock cannot be accurately set by the application. If the clock is stopped when this function is called, the least significant byte of the clock is set to zero. If the clock is running, the value of the least significant byte is undefined. These restrictions mean that, for practical purposes, the clock can be set to a given time with an accuracy of +/- 20 milliseconds while it is running.

1 (miStartClock) Start time stamp clock. This function allocates a DOC generator, writes consecutive values ($01 through $FF) into the first 256 bytes of DOC RAM, and starts the clock. By default, the time stamp clock initially starts at zero. If it is stopped and restarted, it continues at the current value unless the *miSetClock* function is executed to set a new value. The low byte of the time stamp clock always starts at $01, regardless of its previous setting.

We recommend that the clock be started before the MIDI input process is started (*miStartInput*). *miStartClock* performs many time-consuming operations with interrupts disabled. Therefore, data lossage can occur if high-speed MIDI data is being received at the time the *miStartClock* function is called.

NOTE: The Sound Tools and Note Synthesizer must be started prior to this call.

Possible errors:
$0810: no DOC chip or DOC RAM found.
$1921: no DOC generator could be allocated.

$1923: Note Synthesizer isn't started

2 (miStopClock) Stop time stamp clock. The DOC generator and DOC RAM used to implement the time stamp clock are released, and the clock is stopped. Any MIDI data received while the clock is stopped will be tagged with time stamps containing the static value of the stopped clock (the least significant byte will be zero). Output packets with time stamps greater than the static value of the clock will not be output until the clock is restarted or set to a lower value.

3 (miSetFreq) Set clock frequency. This call is important for compatibility with MIDI Tool Set implementations on future machines. Although the frequency of the clock on the Apple //GS cannot be changed, this call will allow the clock frequency to be changed on future machines. Since the clock frequency affects the rate of playback on output, the clock frequency used for playing a sequence should be the same as the frequency of the clock used to record the sequence, or else the time stamps will have to be adjusted to take into account the differing clock rates.

The long parameter contains the number of clock ticks per second. If the value is zero, the clock is set to its default frequency. On the Apple //GS, the only legal choices are 13160 (decimal) or zero (which defaults to 13160).

Possible errors:

miBadFreqErr ($2009): The clock could not be set to the requested frequency. Use *MidiInfo* to find what frequency the clock was set to. If the difference between the actual clock rate and the requested clock rate is too great, time stamps will have to be adjusted arithmetically.

Note: see function *MidiInfo* for methods to read the current clock value and clock frequency.

MidiInfo **function $0C20**

Inputs:
Long - space for result
Word - function number (see below)

Outputs:
Long - the requested information

Errors:
see individual functions below

MidiInfo is the counterpart of *MidiControl*. It returns information about the state of the MIDI Tool Set that an application might need to know.

Function numbers are as follows:

0 (miNextPktLen) Number of bytes in next input packet. This function returns the number of bytes (including four-byte time stamp) in the next packet in the input buffer. If no complete packets are waiting to be read, zero is returned.
Possible errors:
miNoBufErr ($2007): no input buffer has been allocated.

1 (miInputChars) Number of bytes waiting in the input buffer. The returned value includes time stamp and length information (6 bytes per packet), error codes, and sometimes up to 12 bytes of extra space at the end of the buffer. Therefore, it is meant to give only a rough idea of whether the application is keeping up with the input data. If this number becomes large compared to the size of the input buffer, a larger input buffer may be necessary.
Possible errors:
miNoBufErr ($2007): no input buffer has been allocated.

2 (miOutputChars) Number of bytes waiting in the output buffer. The returned value includes time stamp and length information (6 bytes per packet), and sometimes up to 12 bytes of extra space at the end of the buffer. It is therefore meant to give a rough idea of how much data is backed up waiting for output.
Possible errors:
miNoBufErr ($2007): no output buffer has been allocated.

3 (miMaxInChars) Maximum number of bytes in input buffer. This function returns a value indicating the maximum number of bytes that were contained in the input buffer since the last *miMaxInChars* call, or since the input buffer was last flushed. This function is especially useful for gathering statistics on input buffer usage and calibrating input buffer size.

4 (miMaxOutChars) Maximum number of bytes in output buffer. This function returns a value indicating the maximum number of bytes that were contained in the output buffer since the last *miMaxOutChars* call, or since the output buffer was last flushed.

5 (miRecordAddr) Address of packet being recorded by *MidiRecordSeq*. This function is used before processing packets recorded by the *MidiRecordSeq* function. All packets in the application's buffer with lower addresses are available for processing. This function can also be used to estimate how soon *MidiRecordSeq* will terminate by hitting the end of the application's buffer.

THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

6 (miPlayAddr) Address of packet being output by *MidiPlaySeq*. All packets in the application's buffer with lower addresses have been output by *MidiPlaySeq*. This function can be used to estimate how soon *MidiPlaySeq* will terminate by hitting the end of the application's buffer.

THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

7 (miClockValue) Time stamp clock value. The current value of the time stamp clock is returned. The clock value is a positive long value which counts in units of 76 microseconds. If the clock is stopped, the low byte of the clock value will be zero.

8 (miClockFreq) Time stamp clock frequency. The current frequency of the time stamp clock is returned. The frequency is indicated in ticks per second. On the Apple //GS, this frequency is fixed at a rate of 13,160 ticks per second. On future machines, the time stamp clock frequency may be set by an application.

MidiReadPacket **function $0D20**

Inputs:
Word - space for number of characters returned
Long - POINTER to array to hold the returned packet
Word - length of above array

Outputs:
Word - number of characters returned

Errors:
miPacketErr ($2001)
Returned packet is not a legal MIDI command. It is missing one or more bytes. Someone may be disabling interrupts for too long.
miArrayErr ($2002)
User array wasn't big enough to hold the next packet . Try again with a bigger array or discard the packet with *MidiControl*.
miFullBufErr ($2003)
The input buffer overflowed and data was lost. The returned packet may contain no data bytes.
miNoBufErr ($2007)
An input buffer must be allocated.
miDevOverrun ($2083)
The device was not serviced fast enough to avoid lossage of MIDI data. Someone is disabling interrupts for too long. The bytes in the returned packet may not form a complete MIDI command.
miDevNoConnect ($2084)
The device driver lost connection to its hardware interface. The bytes in the returned packet may not form a complete MIDI command.
miReadErr ($2085)
Incoming MIDI data experienced a transmission error. The bytes in the returned packet do not have reliable values.

MidiReadPacket returns the length of a "packet" of MIDI data that it has transferred from the input buffer to the indicated array. A value of zero is returned if no packet is available.

When a packet is returned, the first two bytes of the designated array contain a word indicating the length of the MIDI data including a four-byte time stamp. The next four bytes are a long value containing the time stamp, and the actual MIDI data follows the time stamp. For example, a note on command might look like this:

07 00 24 63 03 00 90 40 5C

The first two bytes indicate the length of the MIDI data plus four bytes for the time stamp. In this case, the MIDI data requires three bytes, so 3 + 4 = 7 (stored in typical fashion, least significant byte first). The next four bytes contain the time stamp $00036324. Static time stamps are supplied if the clock is stopped. The actual MIDI data follows the time stamp.

Note that the result word for the above MIDI packet would be 9. The result word is always two greater than the value contained in the first two bytes of the packet.

The presentation of the actual MIDI data is determined by the current input mode (see the *miSetInMode* function of *MidiControl*). If the current input mode is packet mode (the default), only complete MIDI commands are returned to the application. Unless the packet has been returned with an error, the first byte of the packet is always a MIDI status byte. Even if the status byte was originally omitted during running-status transmission, the MIDI Tool Set supplies the correct status byte so that the application program does not have to keep track of running status or deal with packets of varying lengths. (During output, the MIDI Tool Set reencodes running status unless this optimization is prohibited using the *miOutputStat* function in *MidiControl*.) The MIDI Tool Set also supplies an $F7 byte at the end of MIDI system exclusive packets even if none was transmitted. Real-time MIDI commands are never included in the returned MIDI data (see *miSetRTVec* in *MidiControl* if you wish to process real-time commands).

In raw mode, input data is returned to the application without any of the above interpretations. In this case, MIDI data is split between packets if more than two MIDI character times pass between receipt of sequential MIDI characters, or if 4 MIDI characters have already been gathered into the current packet, or if *MidiReadPacket* requests a packet before one has been completed in accordance with the previous two criteria. Therefore, the longest packet returned in raw mode is 10 characters (2-byte length field plus 4-byte time stamp plus 4 MIDI data bytes). Unlike packet mode, a single MIDI command may span several packets in raw mode. Real-time MIDI commands are included in the returned MIDI data unless a real-time vector has been specified to handle real-time commands (see *miSetRTVec* in *MidiControl*).

The length of the designated array must be at least one byte longer than the length of the packet to be returned (including the 2-byte length indicator and the 4-byte time stamp).

A very rare situation can arise when a continuous stream of large system exclusive packets is received. Occasionally a packet will receive a time stamp that is less than the time stamp of the preceding packet. If this could cause your application difficulty, please design accordingly.

MidiWritePacket **function $0E20**

Inputs:
Word - space for number of bytes written to output buffer
Long - POINTER to array containing a MIDI packet

Outputs:
Word - number of bytes written to output buffer

Errors:
miNoBufErr ($2007)
Output buffer must be allocated first.

MidiWritePacket queues the MIDI packet at the given address for output at a particular time. The format of the packet is the same as that used by *MidiReadPacket* (two bytes of length information, followed by a four-byte time stamp, followed by the actual MIDI data). For example, the following packet would cause a note off command to be output at time $0002537A:

07 00 7A 53 02 00 80 4C 24

The result word indicates the number of bytes written to the output buffer. If this value is zero, the output buffer was full, and the packet could not be written at that particular time. In this case, the application should call *MidiWritePacket* again with the same packet. Eventually, the output buffer will clear (if the output process is enabled), and the packet will be accepted. When the result word is non-zero, it indicates the total length of the packet that was just submitted. For the example given above, this value would be 9. In this example, 0 and 9 are the only possible return values; *MidiWritePacket* does not write only part of a packet.

MidiWritePacket returns to the application within 1/50 of a second, but waits asynchronously until the time stamp clock is greater than or equal to the time stamp contained in the time stamp field of the packet. When the time stamp has expired, it transmits the MIDI data.

The contents of the actual MIDI data depend on the current output mode (see the *miSetOutMode* function in *MidiControl*). In packet mode (the default), the first byte of the MIDI data in every packet submitted to *MidiWritePacket* should be a MIDI status byte (high bit set), and only one complete MIDI command should be contained in a packet. The MIDI Tool Set uses these assumptions to scan the output packets to track note on and note off commands, thus insuring proper operation of the intelligent note off function of *miFlushOutput* in *MidiControl*.

In raw mode, no note tracking is attempted, and therefore there are no restrictions on the contents of each packet. Multiple or partial MIDI commands may be contained in a single packet.

In either mode, status bytes are omitted during output whenever possible using running status optimization, unless this feature is disabled using the *miOutputStat* function in *MidiControl*.

MidiWritePacket may be called before the MIDI output process is actually started (*miStartOutput* function of *MidiControl*). In this case, packets are simply queued in the output buffer until *miStartOutput* is called. It is also not necessary to start the clock in order to use *MidiWritePacket*. If the clock is stopped, all packets with time stamps less than or equal to the static value of the clock are output immediately, and all packets with time stamps greater than the clock value are held in the output buffer until the clock value is started or changed to a greater value using the *miSetClock* function in *MidiClock*.

Two special time stamp values should be noted. Any packet containing a zero time stamp is output immediately, regardless of the state of the clock, when it reaches the head of the output buffer. Any packet containing a negative time stamp (most significant bit set) is interpreted to contain a MIDI "real-time" message. The data contained in this packet is inserted at the head of the output buffer (but behind any previous real-time data) for immediate transmission, thereby preserving the time-sensitive nature of these messages. MIDI real-time messages can be transmitted in the middle of normal MIDI messages.

Output packets that are buffered using calls to *MidiWritePacket* will be merged with output packets originating from *MidiPlaySeq* at the appropriate times. Therefore, it is possible to play a sequence asynchronously and simultaneously generate additional MIDI output using calls to *MidiWritePacket*.

NOTE for Version 1.1: At present, packets are not time-sorted in the output buffer. This means that a packet with a low time stamp must wait behind a packet preceding it in the buffer with a higher time stamp. In future releases, packets will be sorted in increasing order of time stamps to support applications which produce output packets in a non-sequential manner.

MidiRecordSeq **function $0F20**

Inputs:
To be determined

Outputs:
To be determined

Errors:
To be determined

MidiRecordSeq asynchronously records a MIDI sequence directly into an array provided by the application. This eliminates the overhead of performing multiple calls to *MidiReadPacket* and moving data between the input buffer of the MIDI Tool Set and the application's array. This is the only method that can keep up with long streams of continuous high-speed MIDI data.

THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

MidiStopRecord **function $1020**

Inputs:
To be determined

Outputs:
To be determined

Errors:
To be determined

MidiStopRecord halts an asynchronous *MidiRecordSeq* process prematurely (before the end of the application's array is reached).

THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

MidiPlaySeq **function $1120**

Inputs:
To be determined

Outputs:
To be determined

Errors:
To be determined

MidiPlaySeq asynchronously plays a MIDI sequence directly from an array provided by the application. This eliminates the overhead of performing multiple calls to *MidiWritePacket* and moving data between the application's array and the MIDI Tool Set output buffer. This is the only method that can play long streams of continuous MIDI data at full MIDI speed.

THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

MidiStopPlay **function $1220**

Inputs:
To be determined

Outputs:
To be determined

Errors:
To be determined

MidiStopPlay halts an asynchronous *MidiPlaySeq* process prematurely (before the end of the application's array is reached).

THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

MidiConvert **function $1320**

Inputs:
To be determined

Outputs:
To be determined

Errors:
To be determined

MidiConvert converts an array of packets in *MidiReadPacket* and *MidiWritePacket* format (*packet format*) to Standard MIDI File format (*standard format*), and vice versa. *MidiConvert* may also handle conversion of time stamp values between different clock rates.

THIS FUNCTION IS NOT IMPLEMENTED IN VERSION 1.1.

Appendix A
Error Codes

miStartUpErr	$2000	MIDI Tool Set has not been started up
miPacketErr	$2001	incorrect MIDI packet length received
miArrayErr	$2002	an array passed to a MIDI function had an invalid size
miFullBufErr	$2003	MIDI information was discarded due to a buffer overflow
miToolsErr	$2004	required tools were inactive or incorrect version
miOutOffErr	$2005	MIDI output must be turned on before the requested function can work
< no error >	$2006	< no error defined here >
miNoBufErr	$2007	no buffer allocated yet
miDriverErr	$2008	indicated file was not a valid MIDI device driver
miBadFreqErr	$2009	can't set the MIDI clock to the requested frequency
miClockErr	$200A	MIDI clock wrapped to zero
miConflictErr	$200B	two processes competing for MIDI input (*MidiReadPacket* and *MidiRecordSeq*)
miNoDevErr	$200C	no device driver is currently loaded
miDevNotAvail	$2080	MIDI interface is not available
miDevSlotBusy	$2081	requested slot is already in use
miDevBusy	$2082	MIDI interface is already in use

miDevOverrun	$2083	MIDI interface was overrun by incoming MIDI data (not serviced often enough through polling or interrupts)
miDevNoConnect	$2084	no connection to MIDI interface
miDevReadErr	$2085	framing error in received MIDI data (possibly due to bad MIDI cable or connection)
miDevVersion	$2086	ROM version or machine type is incompatible with the MIDI device driver
miDevIntHndlr	$2087	conflicting top-level interrupt handler has been installed

The Note Sequencer ERS

Revision 1.1P
June 2, 1988

WARNINGS, CAUTIONS, SUGGESTIONS, AND IDEAS ABOUT THE NOTE SEQUENCER.

Warning: When actually playing a sequence the Note Sequencer is running at interrupt time. This means that when your error handler and completion routines are called they are also being run at interrupt time with interrupts disabled.

Warning: The Note Sequencer does not support playing notes of semitone zero. The Note Sequencer uses a value of zero semitone to distinguish a filler note.

Caution: The Jump Command has no error checking on boundaries, this means that you could jump right out of your current pattern, this would be bound to cause some sorts of trouble.

Ideas: allnotesoff and clearincrement can be used to stop a sequence that can be restarted with a set increment.

WHO SHOULD USE THIS TOOL?

The Note Sequencer is an interpreter for a simple music programming language designed to play music in the background of running applications. This can be used by any one who wants to play music from a static file, and any one wanting to play music while performing another task that does not disable interrupts.
The Note Sequencer also for easy output of MIDI through the MIDI Tools provided for the IIGS.

WHAT IS THE NOTE SEQUENCER?

The Note Sequencer is a tool kit that interprets a series of commands called a sequence. These commands tell the Note Sequencer which notes to play and when. In addition the Note Sequencer provides some control structures for altering the playback environment.

It is important to note that the Note Sequencer tool kit only supports playback. Creating a new sequence will require an external utility of some type (a wonderful third party project).

The Note Sequencer is designed to run on an interrupt basis in the background of an application and provisions for synchronization have been made. For applications where critical timing is required, there is a mode of operation ("step mode") that allows the application to control sequence playing. This might be necessary if precise synchronization with graphics is required.

The Note Sequencer also supports MIDI, in that you can send the appropriate MIDI commands defined by the sequence (More on this later, and in Appendix E).

Interrupts must be enabled when using the Note Synthesizer and/or the Note Sequencer. Anything that disables interrupts such as reading the disk drives will disrupt the sound being played.

WHAT IS THE NOTE SEQUENCER DEPENDENT ON?

The Note Sequencer depends on the following tools:

The Memory Manager	(Should be started up by the application)
The Tool Locator	(Should be started up by the application)
Sound Tools	(Started up by the Note Sequencer)
The Note Synthesizer	(Started up by the Note Sequencer)
MIDI Tools	(If MIDI is to be used)

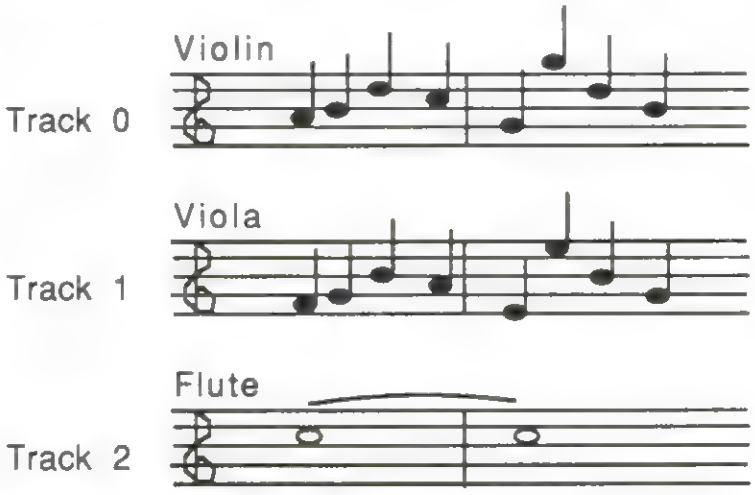
USING MIDI AND THE NOTE SEQUENCER

When using MIDI, the high bit of the Mode parameter in the SeqStartUp call must be set, otherwise MIDI will not be used. If a Sequencer track is used in any way to define the current action, the specified track must also enable MIDI output. Lastly if there is any flag specific to the current command or tool call that must be set for MIDI output this is also checked. If all of these are specified the appropriate, MIDI commands will be sent. This is designed to allow more flexibility in determining when to send MIDI and when not to.

For example, if a sequence is already created for the GS and you want to play it out through MIDI only except for the drum tracks, this can be done easily without changing the sequence. Note: The Midi Tools must be set up by the user, including output buffer, for MIDI to work.

SEQUENCE TIMING:

It is usually convenient to think of a sequence as a series of independent tracks playing at the same time.



The sequence is really a one-dimensional array that is played serially. This means that a chord like this:

is really a series of note on commands like this:

```
play      F4   4  (counts) ; play an F above middle C
play      A4   4  (counts) ; play an A above middle C
play      C5   4  (counts) ; play an octave above middle C
```

If the Note Sequencer waits for the full four counts for each note on command, or SeqItem (Sequence Item). The result will be three separate notes. In order to play the notes as a chord they must all be played with as little delay as possible. This can be done using the chord bit (described later), which enables multiple SeqItems to be performed at the same time.

Every track dependent SeqItem in a sequence contains a track number. Looking at our 3 part example, the sequence for the first measure might be:

	Note	#Ticks	Track	
Play	C5	40	Track 2	(Chord bit set)
Play	A4	5	Track 0	(Chord bit set)
Play	F4	5	Track 1	(delay bit set)
Play	B4	5	Track 0	(Chord bit set)
Play	G4	5	Track 1	(delay bit set)
Play	D5	5	Track 0	(Chord bit set)
Play	B4	5	Track 1	(delay bit set)
Play	C5	5	Track 0	(Chord bit set)
Play	A4	5	Track 1	(delay bit set)

By setting the delay bit on some notes we are indicating that those notes should be completed before the next SeqItem is played. (See Appendix C for an assembly language example of a real sequence).

In the example sequence above we show a quarter note having a duration of 5 ticks. Is this correct? The answer is yes and no. The actual duration of a note is a function of the update rate and the increment. Note that an increment of zero will keep the Note Sequencer from running, but the Note Synthesizer will continue running and maintaining any notes currently being played (see ClearIncr for more details).

The UpDateRate describes how often the Note Sequencer gets interrupts. An UpDateRate of 500 corresponds to a 200 Hz. interrupt ie. an interrupt every 5 milliseconds. Likewise an UpDateRate of 250

corresponds to a 100 Hz. interrupt (10 ms.) This update rate is also used by the Note Synthesizer to update instrument envelopes.

The increment tells the sequencer how many interrupts equal one tick. An increment of 20 and an update rate of 500 would mean that the sequencer would update its counter every 100 ms. That is an interrupt every 5 ms. with the counter incremented every 20 interrupts. Now if each quarter note is 5 ticks that means that each quarter note is 1/2 second long, the equivalent of 120 beats per minute! The following equation shows you how to calculate beats per minute, Where BPM is beats per minute, TicksPB are the number of ticks considered as a beat, and Incr is the Increment supplied in the SeqStartUp call.

BPM = (24 * UpdateRate) / (Incr* TicksPB)

Using faster UpDateRates (or larger numbers) will give the application more control over the tempo as well as providing smoother envelopes for notes. The trade off is that faster UpDateRates require a larger percentage of the processor time.

When assigning durations to notes, you can look at what the shortest note or Filler Note you want to play is. Then give it a duration of 1. From that all other durations can be worked out. For example:

Sixteenth note	= 1
Eighth note	= 2
Quarter note	= 4
Half note	= 8
Whole note	= 16

If the shortest note was to be a 64th note then all of the durations would have to be multiplied by 4. Other changes would of course need to be made for other structures such as triplets.

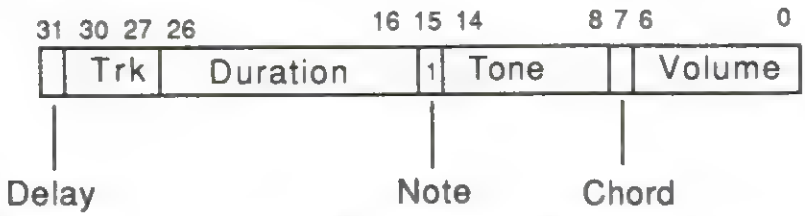
Once an appropriate UpDateRate and note durations have been picked, the only way to vary the tempo is to change the increment. Not only can this value be set at the start of a sequence, but there are control calls that allow the Note Sequencer to change the tempo in the middle of a sequence.

PATTERNS:

A pattern is a series of SeqItems (sequence items). SeqItems are 32-bit commands that describe actions to be performed by the Note Sequencer. There are three different types of SeqItem commands: Note Commands, Control Commands and MIDI Commands. Note Commands turn notes on and off. Control Commands are used to specify such things as pitch bend, program change, sequence looping, and controls other than the actual playing of notes. MIDI Commands allow you to specify the sending of any MIDI Information desirable.

NOTE COMMANDS:

Note Commands are used to start notes playing or to stop notes that are currently playing. The general form of a note command is displayed below.



Bits 0-6 specify the note's volume. This corresponds to MIDI velocity. If the volume is 0 then the player will release any note currently playing with the tone and track specified in the command (Note Off command).

Bit 7 is the chord bit. It is set to indicate that the next sequence item occurs at the same time. This allows the sequence creator to force things like chords which are otherwise timing dependant. Note: If the chord bit is set, the delay bit should not be set in the same command.

Bits 8-14 specify the semitone to be played. The range is 0 to 127 with a value of 60 corresponding to middle C (261.6 Hz).

Bit 15 is always set for note commands. It is always cleared for control commands.

Bits 16-26 specify the duration in timer ticks (0-2047). If the duration is 0, "note on", then the note will continue sounding until the player finds a matching "note off" command (volume of 0). See Appendix D for more information on note on and off commands.

Bits 27-30 Specify the track number. The track number is used as an index to assign instruments and handlers for the note.

Bit 31 is the delay bit. It instructs the player to wait the duration of this command before executing the next command. If this bit is clear then the note is played for the specified duration, but the Note Sequencer will begin playing the next seqitem on the next tick. This allows things such as flams to be played.

Note commands can be used in two ways. Matching pairs of "note on" and "note off" commands can be used in a MIDI-like manner to simulate pressing and releasing keys on a keyboard. if notes are given a duration they will be turned off automatically (no "note off" is necessary), this also will greatly cut down on the size of the sequence. MIDI information can be generated using either form. The values for the semitone and volume fields of the note command are as specified by MIDI with one exception. A note command with a semitone of 0 is interpreted as a "Filler Note".

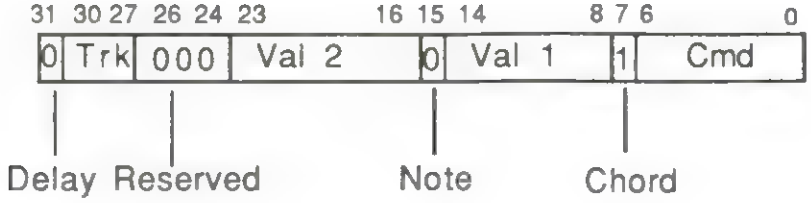
Filler Notes are used primarily to provide timing information to the Note Sequencer. Filler Notes just cause the Note Sequencer to pause. They are treated as a note that can not be heard. See Appendix D for more on Filler Notes, and note on and off commands.

If the delay bit is set then the sequencer will not perform any other action until that event is completed. An example would be making the sequencer hold a chord using the duration format. The first and second notes are played with the delay bit cleared and a certain duration. The third note is played with the delay bit set. The sequencer will not act on any additional seqitems until the third note has completed playing. If the duration for the first two notes is up at this time they will also be silenced, otherwise they will play for their full Duration.
Note: The delay bit is checked only for note commands. Control and MIDI Commands do not have a duration and should never have the delay bit set.

The chord bit tells the sequencer that the next item in the sequence occurs at exactly the same time as the current item. After processing the current item the sequencer will immediately process the next item. This process continues as long as the sequencer encounters items with the chord bit set. The chord bit is particularly useful for playing chords.

CONTROL COMMANDS:

In general, control commands look like the example below. The actual content of many of the fields is command specific.



Bits 0-6 specify a command number.

Bit 7 is the chord bit. It should usually be set, otherwise unwanted delays may be noticed when commands are being processed.

Bits 8-14 contain data specific to the command. This field is referred to as Val 1.

Bit 15 is the note bit. It is always cleared for control commands.

Bits 16-23 contain data specific to the command. This field is referred to as Val 2.

Bits 24-26 are reserved by Apple for future expansion. They should always be cleared.

Bits 27-30 Specify the track number. The track number is used as an index to assign instruments to be used by the track.

Bit 31 is the delay bit. It should always be cleared since control commands don't have a duration.

PitchBend Command

Each Instrument used by the Note Sequencer has a parameter that defines the maximum pitch in semitones that it may be bent. PitchBend is used to raise or lower the pitch within the boundaries defined for that instrument. A PitchBend of 127 causes the maximum raise, and 0 is the maximum for lowering the pitch within the boundaries defined by the instrument. A PitchBend of 64 causes no change. Note: PitchBend is applied to all notes that are currently playing on the track specified in the command. Unlike most synthesizers, subsequent notes played on that track are not bent. The Pitch bend occurs immediately, so many pitch bend commands are needed to make a slurring pitch bend effect. The Reserved parameter is used to set whether the Internal voices are effected, and/or if a MIDI command is generated. If a MIDI is to specified, a MIDI command will only be sent if MIDI is enabled on the specified track.

Cmd = 0.
Chord = 1. Should usually be set to avoid delay in sequence.
Val 1 = Pitch wheel position (0 .. 127, 64 = no pitch bend)
Note = 0. Always cleared for control commands.
Val 2 = not used. Should be 0.
Reserved = 0 : Internal and MIDI; 1 : Internal; 2 : MIDI.
Track = Track Number
Delay = 0. Always cleared for control commands.

Tempo Command

This command will set the Increment to the the contents of Val 1. The old value of Increment is lost.

Cmd = 1.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = new Increment [0 .. 127]
Note = 0. Always cleared for control commands.
Val 2 = not used. Should be 0.
Reserved = 0. Must always be 0.
Track = not used
Delay = 0. Always cleared for control commands.

Turn Notes Off Command

This command will turn off all notes currently being played. It is more compact than sending individual note off commands when a large number of notes must be turned off. This command will also turn off notes turned on by the note synthesizer. At this time it is impossible to support miFlushOutput, in this command, because this command is only executed at interupt time.

Cmd = 2.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = Specifies the low 7 bits for the low word for MIDI Tools miFlushOutput (other bits are cleared), (currently not supported).
Note = 0. Always cleared for control commands.
Val 2 = Specifies the low byte of the High word for MIDI Tools miFlushOutput (high byte is always cleared), (currently not supported).
Reserved =
bit 24 : If set, MIDI Tools will supply a miFlushOutput command as specified by Val 1 and Val 2.
bit 25 : If set, a MIDI AllNotesOff command will be sent (currently not supported).
bit 26 : If set, the Note Sequencer will **not** turn AllNotesOff internally.
Track = not used. Should be 0.
Delay = 0. Always cleared for control commands.

Jump Command

This command works much the same as a jump command in a programming language. The value supplied is the sequencer's program counter. This allows a pattern or a part of a pattern to be repeated indefinitely. Note that the seqitem is referred to by a 15 bit value. The maximum number of items in a individual pattern is 16k. To jump to the beginning of a pattern Val 1 = 0 and Val 2 = 1 (seqitem = 0001). The jump command refers to a particular seqitem rather than a specified number of bytes. Be careful, there is no bounds checking.

Cmd = 3.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = high 7 bits of SeqItem.
Note = 0. Always cleared for control commands.

Val 2 = low 8 bits of SeqItem.
Reserved = 0. Must always be 0.
Track = not used
Delay = 0. Always cleared for control commands.

Vibrato Depth Command

This command will use the contents of Val 1 to change the vibrato depth of all the notes currently playing on the given track. Any subsequent notes will play with the original vibrato depth of the instrument associated with that specified track. The maximum vibrato depth is 127. A value of 0 will turn off vibrato which saves some CPU time. If MIDI is specified, and enabled for the specified track a Control Change Command with a control number specified by bits 16-22 will be supplied with a control value defined by Val 1.

Cmd = 4
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = new vibrato depth [0 .. 127].
Note = 0. Always cleared for control commands.
16-22 = Control number if MIDI command generated.
23 = 0.
Reserved = 0 : Internal and MIDI; 1 : Internal.
Track = Track Number
Delay = 0. Always cleared for control commands.

Program Change Command

This command allows the instrument being used by a particular track to be changed from within the sequence. The instrument selected must already be in the instrument table. If the Reserved area specifies MIDI, and MIDI is enabled for the specified track then a MIDI Program Change command will be generated with the program number specified by Val 2.

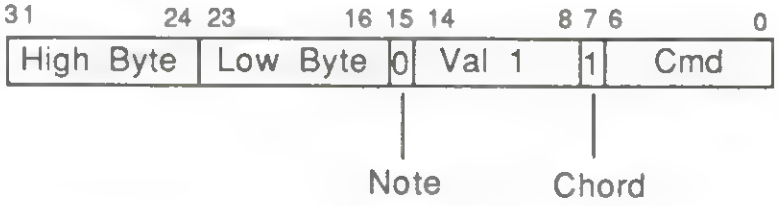
Cmd = 5.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = instrument number (in instrument table)
Note = 0. Always cleared for control commands.
Val 2 = MIDI program number to be used, if using MIDI.
Reserved = 0 : Internal and MIDI; 1 : Internal; 2 : MIDI.
Track = Track Number.
Delay = 0. Always cleared for control commands.

MIDI COMMANDS:

MIDI Commands are provided so that any MIDI data desired can be sent by the current playing Sequence. This set of commands match up with the normal MIDI commands, with the exception of the System Exclusive and the System Common commands which have been replaced by commands that will enable two or one bytes of raw MIDI data to be put out.

Note: the specification of MIDI is not described in this document, but the implementation is based on MIDI Specification 1.0.

MIDI commands have the following format:



MIDINoteOff Command

Sends a MIDI note off command with the channel number specified in Val 1, and the note number by the low byte of the high word, and the velocity by the high byte of the high word.

Cmd = 10.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.
Note = 0. Always cleared for control commands.
LByte = Note number.
HByte = Velocity.

MIDINoteOn Command

Sends a MIDI note on command with the channel number specified in Val 1, and the note number by the low byte of the high word, and the velocity by the high byte of the high word.

Cmd = 11.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.

Note = 0. Always cleared for control commands.
LByte = Note number.
HByte = Velocity.

MIDIPolyphonicKeyPressure Command

Sends a MIDI polyphonic key pressure command with the channel number specified in Val 1, and the note number by the low byte of the high word, and the pressure value by the high byte of the high word.

Cmd = 12.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.
Note = 0. Always cleared for control commands.
LByte = Note number.
HByte = pressure value.

MIDIControlChange Command

Sends a MIDI control change command with the channel number specified in Val 1, and the control number by the low byte of the high word, and the control number by the high byte of the high word.

Cmd = 13.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.
Note = 0. Always cleared for control commands.
LByte = control number.
HByte = control value.

MIDIProgramChange Command

Sends a MIDI note off command with the channel number specified in Val 1, and the program number by the low byte of the high word.

Cmd = 14.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.
Note = 0. Always cleared for control commands.
LByte = Program number.
HByte = 0.

MIDIChannelPressure Command

Sends a MIDI channel pressure command with the channel number specified in Val 1, and the pressure value by the low byte of the high word.

Cmd = 15.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.
Note = 0. Always cleared for control commands.
LByte = Pressure value.
HByte = 0.

MIDIPitchBend Command

Sends a MIDI pitch bend command with the channel number specified in Val 1, and the Pitch Bend Least Significant Byte by the low byte of the high word, and the Pitch Bend Most Significant Byte by the high byte of the high word.

Cmd = 16.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.
Note = 0. Always cleared for control commands.
LByte = Pitch Bend LSB.
HByte = Pitch Bend MSB.

MIDISelectChannelMode Command

Sends a MIDI select channel mode command with the channel number specified in Val 1, and the first data byte by the low byte of the high word, and the second data byte by the high byte of the high word.

Cmd = 17.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = bits 8-11 should be the channel #.
Note = 0. Always cleared for control commands.
LByte = first data byte.
HByte = second data byte.

MIDISystemExlusive Command

This command passes a pointer to a MidiPacket on to the MIDITools so that a MIDI System Exclusive command can be sent from the Note Sequencer. The Low word of the pointer to the MidiArray is supplied by the LByte and HByte parameters for this command. The High Word is supplied by the LByte and HByte of the MIDISetSysExlHighWord Command. When this Command is processed the note sequencer will send the Midi System Exclusive data if MIDI is set in the Mode parameter of the StartSeq Call.
Note: A Note Sequencer MIDISetSysExlHighWord Command Should proceed this one in a Sequence.

Cmd = 18.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = 0.
Note = 0. Always cleared for control commands.
LByte = Low byte of Low Word for pointer to MIDI Array.
HByte = High byte of Low Word for pointer to MIDI Array.

The MIDI Array should look like this:

```
First Word is the Length:    $0008
4 bytes of time stamp:       $0000
  (All zeros for no delay)   $0000
Sys Exclusive command:       $01F0  ; and first data byte.
Next two bytes of data:      $0302
```

MIDISystemCommon Command

Sends a Midi System Common command.

Cmd = 19.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = Bits 8-10 = value 1-7 for low nibble of status byte.
Bits 11-12: determine how many data bytes. If both cleared zero data bytes. Bit 11 set 1 data byte; bit 12 set and 11 cleared, two data bytes.
Note = 0. Always cleared for control commands.
LByte = first data byte, if one.
HByte = Second data byte if one, otherwise zero.

MIDISystemRealTime Command

Sends a system real time command, with the three real time number bits defined by the low byte of the low word.

Cmd = 20.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = 0.
Note = 0. Always cleared for control commands.
LByte = bits 16-18 : real time number.
HByte = 0.

MIDISetSysExlHighWord Command

Sets the high word to be used for a pointer to a MIDI Array for a System Exclusive command.

Cmd = 21.
Chord = 1. Should usually be set to avoid delays in the sequence.
Val 1 = 0.
Note = 0. Always cleared for control commands.
LByte = Low byte of HighWord.
HByte = High byte of HighWord.

REGISTERS:

The sequencer maintains 8 pseudo-registers that can be accessed by both the application and a sequence. Within a sequence there are commands that allow a programmer to create such structures as If..Then, Repeat..Until, Do..While, etc. Since an application can read and write these registers, they provide a means of synchronizing the application and a sequence.

The registers are stored in bytes 2-9 of the sequencer's direct page. The sequencer's direct pages are $100 bytes past the ZeroPageLoc supplied in the SeqStartUp call. The Note Synthesizer and Sound Tools get the first page, while the next two pages are the Sequencer's.

While the registers account for 8 bits in memory, the Set Register and IfGo Register commands only can use the least significant 4 bits of the register. However the Inc Register and Decrement Register Commands will work as if the Registers are a full 8 bits.

Note that while the registers are 1 byte wide, only the bottom 4 bits of them should contain a value. The high nibble should be 0.

The next 4 control commands affect the registers (which are referred to as Registers 0-7).

Set Register Command

Set the specified register to the supplied value.

Cmd = 6
Chord = 1. Should usually be set to avoid delay in sequence.
Val 1 = low 3 bits = register number; high 4 bits = value.
Note = 0. Always cleared for control commands.
Val 2 = not used. Should be 0.
Reserved = 0. Must always be 0.
Track = not used.
Delay = 0. Always cleared for control commands.

IfGo Command

If the specified register has the supplied value (high 4 bits of Val1) then the branch is taken, otherwise no action is taken. Be careful though because there is no bounds checking.

Cmd = 7
Chord = 1. Should usually be set to avoid delay in sequence.
Val 1 = low 3 bits = register number; high 4 bits = value.
Note = 0. Always cleared for control commands.
Val 2 = offset. -128 to +127 seqitems
Reserved = 0. Must always be 0.
Track = not used.
Delay = 0. Always cleared for control commands.

Inc Register Command

Increment the value contained in the specified register.

Cmd = 8
Chord = 1. Should usually be set to avoid delay in sequence.
Val 1 = low 3 bits = register number
Note = 0. Always cleared for control commands.
Val 2 = not used. Should be 0.
Reserved = 0. Must always be 0.
Track = not used.
Delay = 0. Always cleared for control commands.

Dec Register Command

Decrement the value contained in the specified register.

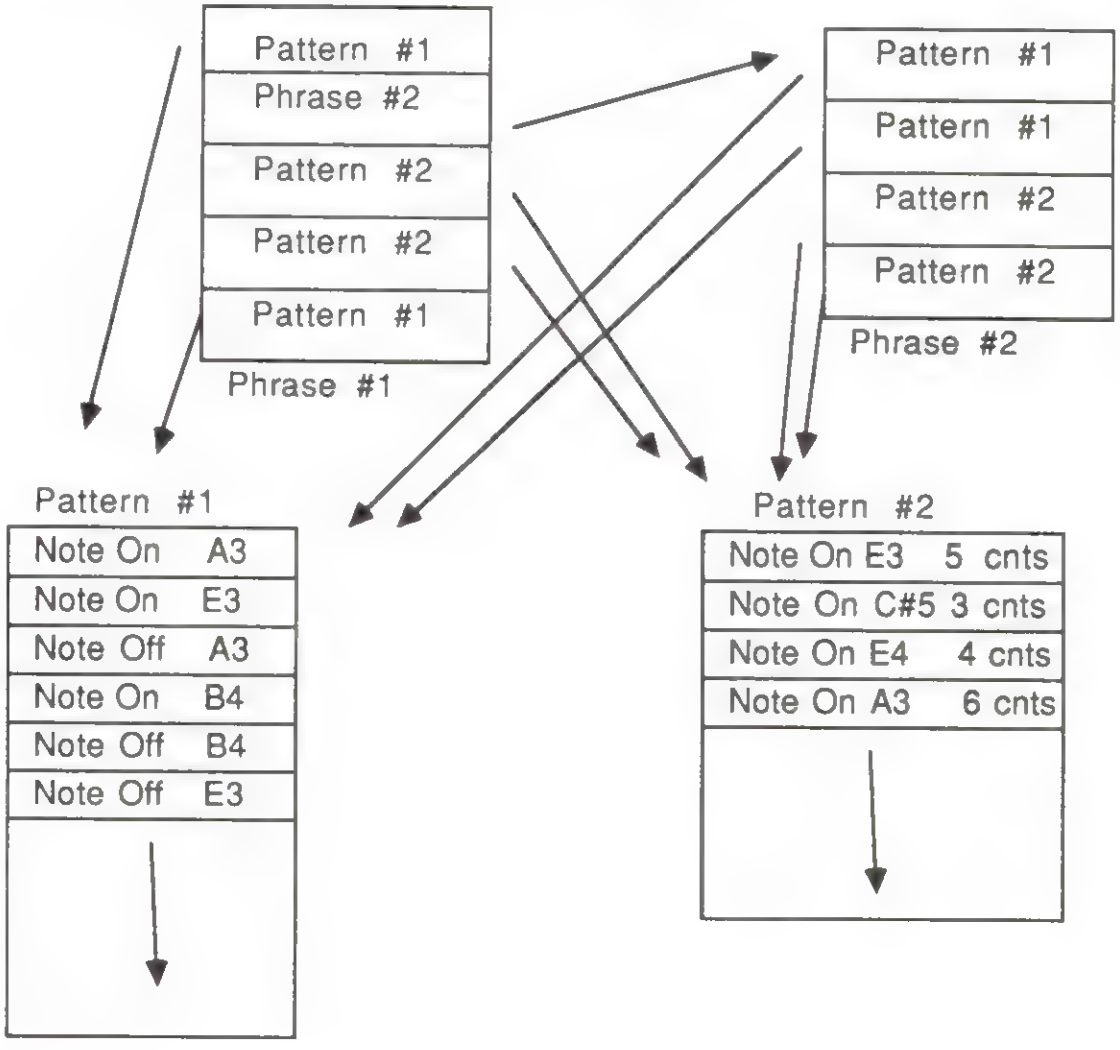
Cmd = 9
Chord = 1. Should usually be set to avoid delay in sequence.
Val 1 = low 3 bits = register number.
Note = 0. Always cleared for control commands.
Val 2 = not used. Should be 0.
Reserved = 0. Must always be 0.
Track = not used.
Delay = 0. Always cleared for control commands.

Dec Register can cause the 8 bit register to wrap from 0 to $FFFF

PHRASES:

A phrase is composed of pointers to patterns and/or other phrases. The term "Sequence" refers to the top level phrase.
The Note Sequencer provides no support for creating new phrases or patterns. Instead the data structures are described completely in Appendix A.

The following example shows a phrase that contains 2 different patterns and another phrase. Pattern 1 uses MIDI style note commands. Pattern 2 uses durations. As shown in the example, each pattern only needs to be stored once. Phrase 2 is not shown.



See Appendix C for an Assembly Language example of a Sequence

THE TOOL CALLS:

All of the tool calls are stack based calls. The Caller is responsible for supplying the extra space needed for output parameters.

SeqBootInit Call Number 1
Parameters: None
Errors: None

Does nothing. All variable initialization, memory allocation, etc. is performed in the startup call.

SeqStartUp Call Number 2

Parameters:

Input	ZeroPageLoc	WORD
	Mode	WORD
	UpdateRate	WORD
	Increment	WORD

Errors:
StartedErr = $1A03
SeqNSWrngVer = $1A07
SoundStartUp Errors
NSStartUp Errors

Sound Tools and the Note Synthesizer should not be started up before this call, because SeqStartUp makes both a SoundStartUp and a NSStartUp call. The Note Sequencer needs 3 pages of bank zero for its direct page starting at the specified address (They should also be locked). The first page is used for the Note Synthesizer and the others are used for the Note Sequencer. Mode is the operation mode for the sequencer. It should be 0 for interrupt mode (ASYNC) or 1 for step mode (SYNC). If Midi is to be used, then the high bit should be set ($8000 for async with MIDI, and $8001 for Sync with MIDI). UpdateRate is the update rate to be used by the Note Synthesizer. The value is specified in units of .4hz. Typical values would be:

Rate: 60 Hz	use 60/.4 = 150
Rate: 100Hz	use 100/.4 = 250
Rate: 200Hz	use 200/.4 = 500 (default if zero)

Low rates should be used for low overhead. Higher rates will give better timing resolution and smoother sounding envelopes.
NOTE: The Note Synthesizer begins generating interrupts at the UpdateRate when the SeqStartUp call is made. In order to speed up applications, the Note Sequencer disables the update interrupt when it is not playing. The timer interrupt is enabled when the application calls StartSeq. The interrupt is disabled when the top-level phrase completes. See Appendix B if you want to use the Note Synthesizer when the Note Sequencer is started up.
If there is an error that keeps the Note Synthesizer from being started up, the SoundTools will be shut down automatically.

Increment is the number of interrupts to count before a Tick occurs. This value controls the tempo of the piece being played. Small values will play the piece faster than large values. The following equation

can be helpful in calculating timing, where TicksPB is the number of ticks that is to be considered a beat, and BPM is Beats Per Minute.:

Increment = (update rate* 24)/(BPM * TicksPB)

SeqShutdown Call Number 3
Parameters: None
Errors: NotInit = $1923 from Note Synthesizer
NoStartErr = $1A04

This routine frees any buffers that might have been allocated by the Note Sequencer. It should be called by an application before quitting.

SeqVersion Call Number 4
Parameters:
output VersionInfo WORD
Errors: None

Returns the version of the Note Sequencer.

SeqReset Call Number 5
Parameters: None
Errors: None

This routine is called by the system when a reset is performed. All internal notes are turned off.

SeqStatus Call Number 6
Parameters:
output StatusFlag Boolean
Errors: None

ActiveFlag is non zero if the Note Sequencer is active and zero otherwise.

(reserved) Call Number 7

This call number is reserved for future system enhancements.

(reserved) Call Number 8

This call number is reserved for future system enhancements.

SetIncr Call Number 9
Parameters:
input Increment WORD
Errors: None

Allows the application to supply a new value for Increment.
Note: Increment may also be changed from within a sequence.
Note: This call is only valid if a sequence is playing.
Note: If Increment is zero, the sequencer doesn't progress (See ClearIncr).

ClearIncr Call Number 10
Parameters:
output Increment WORD
Errors: None

Sets Increment to 0 and returns the old value of Increment. This stops the Note Sequencer's timer from being incremented. The timer interrupt is still active so envelopes are updated as necessary. However, Durations are not checked, so this may result in notes being hung until the increment is set to a non-zero value.
Note: This call is only valid if a sequence is playing.

GetTimer Call Number 11
Parameters:
output CurrentTime WORD
Errors: None

Returns the current value of the sequence timer. The Timer is just a tick counter.
Note: If the timer has not been stopped, the result is probably wrong.
Note: This call is only valid if a sequence is playing.

GetLoc Call Number 12

Parameters:

output	CurPhraseItem	WORD
	CurPattItem	WORD
	CurLevel	WORD

Errors: None

The information needed to compute where the sequencer is at, is provided. CurPhraseItem is the current pattern in the active phrase. CurPattItem if non-zero, is the SeqItem number being played in the current pattern. Level is the number of phrases deep we are. If Level is zero then we are in the top level phrase.
Note: This call is only valid if a sequence is playing.

Example:

Top Phrase ->
Phrase 1, Phrase 2
Phrase 1 -> Pattern 1

If we are playing the third item in Pattern 1. The results of GetLoc should be CurPattern = 1, CurSeqItem = 3, Level = 1.
Note: This is a good call to make in the ErrHndlrRoutine.

SeqAllNotesOff Call Number 13

Parameters: None

Errors: None

Almost the same as the Note Synthesizer call of the same name. This call turns off all the notes currently playing, it does not stop the Sequence from playing. Note: Some additional cleanup is done by this call. If you want to turn off notes while the Sequencer is active, use this call! If the high bit of Mode is set in the SeqStartUp call then all of the MIDI notes the sequencer knows of will be turned off.

SetTrkInfo Call Number 14

Parameters:

input	Priority	WORD
	InstIndex	WORD
	Tracknum	WORD

Errors: InstBoundsErr= $1A01

Sets the information to be used by a track. This call should be made prior playing a sequence, and for each track used. Priority is the priority value used when allocating generators. InstIndex specifies which instrument in the instrument table to use for the track specified by TrackNum. If the MSB (most significant bit) of the InstIndex is set, no internal voices will be played for that track. If the MSB of TrackNum is set and the Mode specifies using MIDI, then everything played on that track will produce MIDI information on the channel number specified by the second most significant byte of TrackNum. For example, a TrackNum of $8201 would mean that everything played by track one would produce MIDI information on MIDI Channel two. A SetInstTabl call must be made prior to any SetTrkInfo Calls.

StartSeq Call Number 15

Parameters:

input	ErrHndlrRoutine	LONG
	CompRoutine	LONG
	Sequence	HANDLE
Errors:	miNoBufErr = $2007	
	NoStartErr = $1A05	

Errors returned to ErrHndlrRoutine:

NoRoomMidiErr = $1A00
NoCommandErr = $1A01
NoRoomErr = $1A02
NoNoteErr = $1A04 Couldn't find note for note off.
AlreadyOn = $1924 from Note Synth.
NoneAvailable = $1921 from Note Synth.
miToolsErr = $2004 From Midi Tool Set

Will Start a Sequence. Sequence must be a handle to a phrase. If the sequencer is in interrupt mode, the sequence will play until complete. If the sequencer is in step mode then StepSeq should be called repeatedly to play the sequence. When the sequence has completed the sequencer will JSL to the address specified by CompRoutine (if non-zero). In any event the done flag will be set to $FFFF when the sequence has completed. The done flag is the first word in the sequencer's WAP. (ie. $100 + ZeroPageLoc). The application may poll this location. If the StartSeq has been called and the sequence has not been completed then the done flag will be set to 0. ErrHndlrRoutine is a pointer to the users error handling routine. A default of zero for the ErrHndlrRoutine will mean that it will be ignored. Otherwise the routine pointed to will be called by The Note Sequencer when an error occurs while playing a sequence. If The Note Sequencer is in interrupt mode, the error in the sequence will only be reported to the ErrHndlrRoutine (if supplied), but if it is in step mode the errors will also be returned in the accumulator after the StepMode call.
Note: Note Synth Interupts are disabled when the user routines are called. And the DataBank used when _SeqStartUp was made is supplied in both the ErrHndlrRoutine and the CompRoutine. The error code will be passed in the accumulator. One good function of this routine could be to set a flag that the completion routine will look at saying that an error occurred, and to do a Note Sequencer _GetLoc call to remember the location that the error occurred at.

Note: The Sequencer calls the ErrHndlrRoutine and Completion Routine with it's direct page in the direct page regester. So, if you want your own dp you will have to load it your self, but you will not need to restore the dp regester.
Note: Sequence should be a locked handle!
Note: ErrHndlrRoutine and CompRoutine are executed with stack pointer in $0001XX area and care should be taken not to put a lot of baggage on the stack or a crash is likely. This is because the Note Sequencer executes at an error handling level. This also means you shouldn't spend a lot of time in either routines.
For Example: a C printf call will push to much stuff on the stack in most cases.
Note: If an error is returned from this call the Note Synthsizers oscilator is not forced on.

StepSeq Call Number 16
Parameters: None
Errors: NoRoomErr = $1A02
NoNoteErr = $1A04 Couldn't find note for note off.
AlreadyOn = $1924 from Note Synth.
NoneAvailable = $1921 from Note Synth.
NoCommandErr = $1A01

Increments the sequence timer. Plays the next item in the current sequence if necessary. The current sequence must be set by calling StartSeq prior to making this call with the mode set to StepMode (1). Control is returned to the application when the current item has been played.

StopSeq Call Number 17
Parameters:
input Next Boolean
Errors: None

Stops the sequence that is currently being played. If Next is non-zero, the timer is running and there are additional phrases in the sequence, the next phrase will begin playing. Otherwise the sequencer will simply stop. The completion routine is called if Next is zero.
All Notes are turned off when this call is made, including Note Synthesizer Notes and Midi Notes generated by the Note Sequencer.

SetInstTable Call Number 18
Parameters:
input InstTable HANDLE
Errors: None

InstTable is a handle to an array of 4 byte pointers to instruments, with a one word header. The one word header is the number of instruments in the table. Within the sequencer, instruments are referred to by their number in the table (0,1, . . . n). The table looks like:

NumOfInstrmnts
Pointer to Inst 0
Pointer to Inst 1
.
.
.
Pointer to Inst n

NumOfInstrmnts in this case would be n+1.
Note: This call must be made prior to the SetTrkInfo call (See "Apple IIGS Sampled Instrument Format" for specification on instruments).

StartInts Call Number 19
Parameters: None
Errors: None

This call will restart the Note Synthesizer's timer interrupt.

StopInts Call Number 20
Parameters: None
Errors: None

This call will halt the Note Synthesizer's timer interrupt. An application should make this call if they do not want interupts going off when not playing a sequence. Ideally this call would be made after the SeqStartUp call and before the startseq call. That way interupts aren't being generated untill the startseq call is made. This call should be made only when no notes are playing by the Note Sequencer or the Note Synthesizer. The advantage to this call is that unwanted interupts can be avoided by making it. Use the

StartInts call to restart the Note Synthesizer's timer interrupt. The StartSeq call does this automatically.

Appendix A: Data Structures for Patterns and Phrases

Both patterns and phrases are arrays of long words that end with the tag $FFFFFFFF. They are differentiated by their 4 byte headers.

Phrase:

The declaration for the header of a phrase is :

```
Phrase    dc    i2'0001'        ; low word
          dc    i2'0000'        ; high word
```

This is followed by pointers to other phrases or patterns.

```
          dc    i4'phrase1'
          dc    i4'pattern'
```

The phrase must end with the terminator.

```
          dc    i4'$FFFFFFFF'
```

Patterns:

The declaration for the header of a pattern is :

```
Pattern   dc    i2'0000'        ; low word
          dc    i2'0000'        ; high word
```

This is followed by sequence items, such as:

```
C         dc    i4'$880ABC74'
D         dc    i4'$880ABE74'
E         dc    i4'$880AC074'
```

The above seqitems play C4, D4, and E4 with dur = 10 and Vol = 115.

The pattern must end with the terminator:

```
          dc    i4'$FFFFFFFF'
```

Note: The Sequencer supports 14 levels of phrase nesting.

Appendix B: Using The Note Synthesizer with the Sequencer

When the Note Sequencer is started it tells the Note Synthesizer to call it's interupt handler when ever there is an interupt (assuming that the low bit in Mode is cleared). This is useless over head if you aren't playing any notes. So the oscilator used to generate this interupt can be halted by a StopInts call. This will stop the interupt, the Note Sequencer and will keep any Note Synthesizer notes from being updated, includeing their envelpes. To restart the oscilator and the interupts make a StartInts call.

The Note Sequencer only starts the interupt automatically when it is started up and during a StartSeq call. It only turns it off automatically when it does a SeqShutDown call. Otherwize the interupt oscilator is never altered except by the StopInts and StartInts calls.

Note that there are some Note Synthesizer calls such as AllNotesOff that shouldn't be made. Other calls may effect how the Note Sequencer keeps track of what it's playing. For example: A Note Off call to the Note Synthesizer may turn off a note being played by the Note Sequencer.

Appendix C: An Assembly Language Example of a Sequence.

```
Delay       equ  $80000000
T1          equ  $08000000
T2          equ  $18000000
qtr         equ  $40000
hlf         equ  $80000
Note        equ  $8000
C4          equ  $3C00
D4          equ  $3E00
F4          equ  $4100
G4          equ  $4300
Chord       equ  $80

phrhndl     dc   i4'phr1'
phr1        dc   i4'01'            ;it's a phrase
            dc   i4'phr2'
            dc   i4'pat1'
            dc   i4'pat2'
            dc   i4'$FFFFFFFF'     ;end of phr1

phr2        dc   i4'01'            ;it's a phrase
            dc   i4'pat2'
            dc   i4'pat1'
            dc   i4'$FFFFFFFF'     ;end of phr2

pat1        dc   i4'00'            ;it's a pattern
            dc   i4'Delay+T1+qtr+Note+C4+115'
            dc   i4'T1+qtr+Note+C4+Chord+115'
            dc   i4'Delay+T2+qtr+Note+G4+115'
            dc   i4'Delay+T1+hlf+Note+F4+115'
            dc   i4'$FFFFFFFF'     ;end of pat1

pat2        dc   i4'00'            ;it's a pattern
            dc   i4'T1+Note+G4+Chord+115'  ;Note On
            dc   i4'Note+hlf'      ;Filler Note
            dc   i4'Delay+T2+qtr+Note+F4+115'
            dc   i4'Delay+T2+qtr+Note+D4+115'
            dc   i4'T1+Note+G4+Chord+115'  ;Note Off
            dc   i4'$00000002'             ; All notes off.
            dc   i4'$FFFFFFFF'             ;end of pat1
```

Appendix D: Filler Notes, and Note On and Off Commands.

Filler Notes are notes that do not make any sound but have duration. This is useful for controlling timing problems that may arise when using Note On and Off commands, or when playing chords. An example would be playing a chord that has several notes of different length, followed by a run of other notes. You may want to have a Filler Note as the last note of the chord, so that you can easily vary the amount of delay between when the chord is struck and when the run starts.

Filler Note

Delay:	should be set if a delay is wanted.
Trk:	Not Used.
Duration:	Desired delay time.
Note:	1. Must be set.
Tone:	0.
Chord:	Should be set.
Volume:	Not Used.

NoteOn

Delay:	0
Trk:	0-15
Duration:	0
Note:	1
Tone:	Any
Chord:	Set as Desired
Volume:	Any

NoteOff

Delay:	0
Trk:	0-15 same as matching note on.
Duration:	0
Note:	1
Tone:	Same as matching note on.
Chord:	Set as Desired
Volume:	0

Appendix E: MIDI and the Note Sequencer.

In order to use MIDI with the Note Sequencer, the appropriate MIDI calls must be made to set up the MIDI Tools (See the MIDI Tool Set ERS). Once the MIDI Tools are ready to start output the sequence can be played and will put out the appropriate MIDI data given that the Note Sequencer and the Sequence are set up properly and are prepared to send MIDI Data.

Note: The necessary requirements for MIDI output are:

1). Start up the MIDI tools set.

2). Set a device driver.

3). enable output to be started. (Since the Note Sequencer sends midi data immediately with no time stamps, a small buffer is all that is needed).

To get the Note Sequencer set up properly for MIDI Data the SeqStartUp call must be made with the high bit of the Mode parameter set. This will enable the use of MIDI in the by the Note Sequencer. Next, each track that is to support MIDI must have a SetTrkInfo call for it, with the highest bit of the parameter Tracknum set to enable MIDI, and the third byte (low byte of high word) of Tracknum will specify which MIDI channel number will be used for the MIDI information put out by that track. Once all this has been done, the Sequencer will generate MIDI data as it plays the Sequence (unless specified other wise).

Appendix F: Note Sequencer Errors.

$2007	miNoBufErr	Returned if MIDI is specified in Mode parameter of SeqStartUp call and the StartSeq call is made without allocating an output buffer for the MIDI Tools.
$1A00	NoRoomMidiErr	There was no room for the MIDI Note On. The Sequencer is already keeping track of 32 notes that are still playing.
$1A01	NoCommandErr	The Note Sequencer can't understand the current SeqItem in the context.
$1A02	NoRoomErr	The Sequence is more than twelve levels deep.
$1A03	StartedErr	The Note Sequencer is already Started Up.
$1A04	NoNoteErr	Can not find the note that is to be turned off by the current SeqItem.
$1A05	NoStartErr	The Note Sequencer hasn't been Started yet.
$1A06	InstBndsErr	The Instrument Boundaries don't allow for the specified instrument number.
$1A07	SeqNSWrngVer	Note Synthesizer Wrong Version. The Note Synthesizer is an incompatible version with the Sequencer. Try upgrading to the latest Note Synthesizer.